Parallel Methods for Single-Row Routing Problem

A thesis submitted
in partial fulfillment
of the requirements
for the degree of

Master of Technology

by

SAIBAL DAS

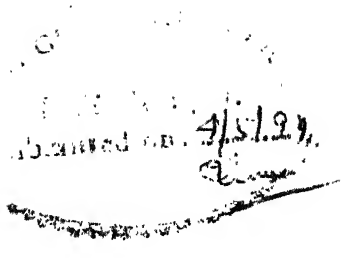
to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

April,1994

CERTIFICATE

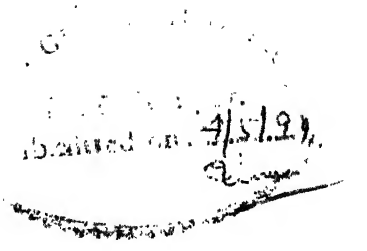


This is to certify that the work contained in the thesis titled, **Parallel Methods for Single-Row Routing Problem**, was carried out under my supervision by **Saibal Das** and it has not been submitted elsewhere for a degree.

Dr Sanjeev Saxena
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur April 1994

E-1994-M-DAS-PAR

CERTIFICATE



This is to certify that the work contained in the thesis titled, **Parallel Methods for Single-Row Routing Problem**, was carried out under my supervision by **Saibal Das** and it has not been submitted elsewhere for a degree.

Dr Sanjeev Saxena
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur April 1994

Thanks ...

to my guide, Dr Sanjeev Saxena, who has been an inestimable help through the course of my thesis. His constructive comments and unerring instinct in pointing out the weaknesses in my arguments helped me to form a clearer understanding and chart out a better course of action. I am greatly indebted to him for enriching my knowledge.

to Dr Gautam Barua, Dr S.C.Srivastava and Dr A.Mahanta for attending my presentation and expressing their views.

to Sanjay, Tiwari, Dhir, Wadhwani, Tribhu, Pradeep and Meena whose friendship during my B.Tech days will always keep the memory of i.i.t.k. alive in my mind.

to Banerjee, Uzzal, Bairagi, Vinod, Shomi, Tan, Rao, Malhal and others who made my stay here during the last two years an enjoyable one.

to my teachers, they have played an important role in shaping my future. I'll never be able to forget the care and concern they have shown.

to my parents for their faith and confidence in me and who will always be there to encourage me in every venture and adventure I would like to undertake.

SAIBAL

ABSTRACT

In this thesis special cases and heuristics for Single-Row Routing Problem on Parallel Computers are discussed.

Single Row Routing Problem for the cases when the number of tracks available are atmost 2, $O(\log n)$ time algorithms with $\frac{n}{\log n}$ processors on a Concurrent Read Exclusive Write computer are obtained. These algorithms can also be implemented on a *Tree Machine* in $O(\log^2 n)$ time with $\frac{n}{\log^2 n}$ processors.

For the general case, two parallel heuristics are proposed. Experimentally, the performance of these heuristics is comparable to the earlier sequential heuristics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the design of electronic systems, the design of multilayer printed circuit boards (MPCB's) is of great importance.

According to So [So 74] [Raghavan 83], the design and layout of MPCB involves:

1) placement of the functional modules of the system on the MPCB, and

2) conductor routing, subject to various physical constraints, on the MPCB to effect the necessary interconnections between the modules.

These two problems are generally treated separately because of the total layout problem complexity.

We shall mainly be concerned about the second step. Given the module locations on the MPCB and the lists of points to be made electrically common, we have to find the conductor paths on the MPCB that satisfy all the requirements and constraints.

MPCB generally tends to have a regular geometry. The points to be connected are uniformally spaced on a rectangular grid. For such cases, So [So 74] [Raghavan 83] has proposed a systematic decomposition of the general multilayer routing problem into a number of independent single-layer, single-row routing problems. Following this decomposition, there is one single-row routing problem for every horizontal and every vertical lines of points in the original problem. All these single-row problems are solved independently of one another. The available routing layer are divided into disjoint sets. The solutions to all the 'horizontal' single-row problems are placed on one set of layers, and the solutions to the 'vertical' problems on the other set. So[So 74] [Raghavan 83] called this approach unidirectional routing. The major advantages of this approach are as follows [So 74] [Raghavan 83]:

1) It allows efficient *'a priori'* estimation of the inherent routability of an given instance whereas other popular routing methods like the maze routers [Breuer 72], channel routers [Hashimoto 71] etc. lack this prognostic capability.
2) Single-Row Routing produces wire layouts that are more amenable to automated fabrication, since all conducters on a given layer are either horizontal or vertical.

1.1 The Single Row Routing Problem

Single-row routing problem is defined by a set of n nodes placed on a row and a net list $L = \{N_1, N_2, N_3, \ldots, N_m\}$ which prescribes the connection pattern of the nets to the nodes. Each net of the net list is to be realised by horizontal and vertical paths connecting the nodes according to specification and there should not be any overlapping between nets. The space above the row R is referred to as *the upper street* and the space below R is referred to as *the lower street.* The number of horizontal tracks allowed in the upper street is called the *upper street capacity*, and number of horizontal tracks actually used in a realization in the upper street is called the *upper street congestion. Lower street capacity* and *congestion* are similarly defined. The following notation is used:

- C_{us} : upper street congestion
- C_{ls} : lower street congestion
- c_{ui} : number of horizontal tracks passing above the i^{th} node in the realization
- c_{li} : number of horizontal tracks passing below the i^{th} node in the realization

From the definition:
C_{ls} : max$\{ c_{li} \mid i : 1, \ldots, n\}$
C_{us} : max$\{ c_{ui} \mid i : 1, \ldots, n\}$

An optimum realization is the one which minimises the street congestion $Q_o = \max\{C_{us}, C_{ls}\}$. We say that an optimum realization is achieved if Q_o is minimum.

It has been shown in [Kuh 79] that to each realization of a given net list L, there corresponds a unique Interval Graphical Representation. The Interval Graphical Representation is a set of m horizontal intervals representing the m

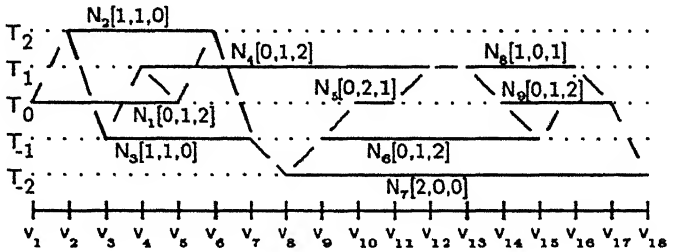


Figure 1.1: Interval Graphical Representation–Single Row realization is described in Figure 1.2

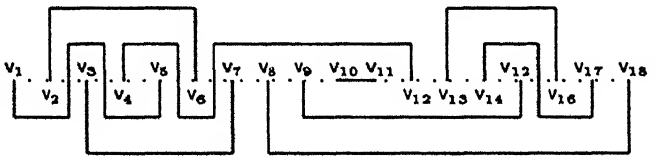


Figure 1.2: Single-Row realization of Interval Graphical Representation of Fig 1.1

nets together with an assumed order. Figure 1.2 shows the single-row realization of the Interval Graphical Representation of Figure 1.1, obtained by topological mapping of the space, above and below the broken line when the broken line is stretched to lie on the straight single row.

- Definition 1: The cut number c_i at node i is the number of nets cut by the vertical line superimposed on the Interval Graphical representation at the i^{th} node v_i; From the previous definition:

  $c_i = c_{ui} + c_{li}$

- Definition 2: Cut number of a net N_j denoted by q_j is the maximum over cut numbers of the nodes which belongs to the net N_j.

After Tarng et.al. [Tarng 84] we classify nets according to their cut numbers and introduce the concept of zone as follows: We assign nets with the maximum cut numbers q_m to the zone Z_0, and the nets with cut numbers $q_m - i$ are assigned to zone Z_i.

- Definition 3: Given a net list $L = \{N_1, N_2, \ldots, N_m\}$ and a division of L into two subsets $L1$ and $L2$ such that $L1 \cap L2 = \phi$ and $L1 \cup L2 = L$. Let $N_i \in L1$. The internal cut number iq_i of the net N_i in L with respect to $L1$ is defined as the cut number of N_i in $L1$.

- Definition 4: The residual cut number rq_i of the net N_i in L with respect to $L1$ is defined as cut number of N_i in $\{L2 \cup N_i\}$.

In addition to the optimum realization in which one minimizes the *street congestion*, it is sometime desirable to minimize the vertical tracks between two nodes. This problem was addressed to by Du et al. in [Du 87]. Here, instead of just dealing with the number of tracks needed for the realization of a set of nets, we also take into account the number of cross-overs (between adjacent nodes) needed for the layout, our measure of optimality becomes the total area needed for the realization. The number of vertical tracks available between any two consecutive nodes is called between nodes capacity or vertical track capacity. We refer to it as the *Cross-over Bound*. The objective of our problem changes when we take into account the *Cross-over Bound* which now is to minimize the total area, A, given by
$A = (K + 1) * (n - 1) * (M_u + M_l)$
where K is the maximum number of *Cross-over* between any two adjacent nodes, n is the number of nodes and M_u and M_l are the maximum number of tracks needed on the upper and lower streets, respectively.

1.2 Structure of the Report

In Chapter 2, we introduce various Parallel Models of Computation used in this thesis.

In Chapter 3, we parallelize the Tsukiyama et al. algorithm [Tsukiyama 80] for Single-Row Routing with Prescribed Street Congestion. Later, in same Chapter, we present an algorithm on Single-Row Routing with Prescibed Street Congestion and Cross-Over Bound by using an approach similar to that used by Du et al. in [Du 87].

In Chapter 4, we discuss the implementation of the algorithm of Chapter 3 on a *Tree Machine*.

In Chapter 5, we parallelize the heuristic due to Tarng et al. [Tarng 84] using two approaches. In the first approach, we present another $O(log\ n)$ parallel heuristic; this heuristic takes $O(log\ n)$ parallel time. We compare the success rate of the proposed heuristic by comparing the heuristic with that of Tarng et al. heuristic [Tarng 84]. In the second approach, we present an $O(t\ log\ n)$ time parallel implementation of the heuristic due to Tarng et al. [Tarng 84].

In Chapter 6, conclusions are drawn and scope for further work are suggested.

Chapter 2

Parallel Models of Computations

Parallel Random Access Machine (PRAM) model is perhaps the most popular model of parallel computers for algorithmic design. It neglects any hardware constraints and is ideal for studying inherent parallelism present in a problem. It gives absolute freedom to algorithm designer in presentation of parallel algorithms. In any realization of PRAM there will be a link between each processor and each memory location. This is however not realizable with the present day architectures. There are methods of simulating (like sorting on address and processor index) such an idealized computer on more reasonable parallel computers (like fixed networks of processors with number of linkages from any processor being bounded). This simulation costs only polylogarithmic time on most models [Alt 86].

PRAM is a shared memory model. Many processors work synchronously and communicate through the common random access memory. Each processor is a uniform cost Random Access Machine(RAM), with usual operations and instructions (see e.g., [Aho 74] for details of RAM model).

The processors are indexed by consecutive integers usually from the number 0 (i.e. 0 - (n-1) if there are n processors).

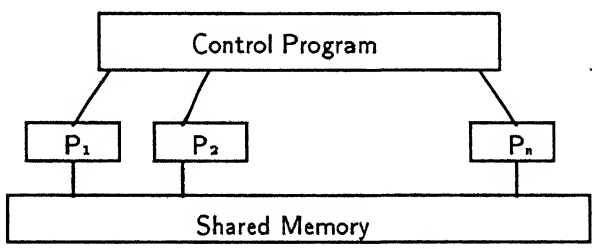


Figure 2.1: The Parallel Model of Computation

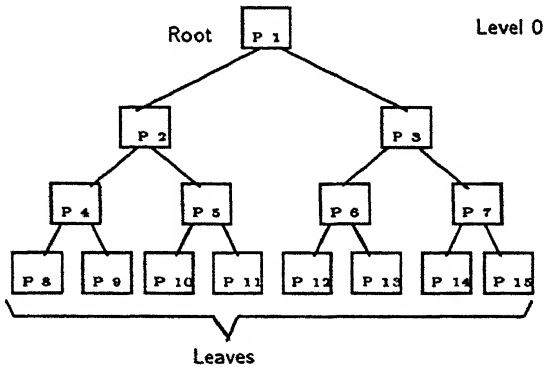


Figure 2.2: Tree Machine

See Figure 2.1. All processors execute the same instructions but (possibly) on different data. Hence it is a single instruction multiple data (SIMD) stream model.

In one step each processor either reads or writes into a memory location. There are three models in PRAM family depending on rules for the simultaneous access of the same memory location.

EREW PRAM EREW is a acronym for Exclusive Read and Exclusive Write. In this model no two processors can attempt to read the same memory location at the same time; nor two processors can attempt to write into the same memory locations at the same time. This is the weakest model among all PRAM models.

CREW PRAM This stands for Concurrent Read and Exclusive Write. In this model two or more processors can read the same memory location at the same time, but still no two processors can try to write into the same memory location simultaneously.

CRCW PRAM In this model, simultaneous concurrent reads and concurrent writes are permitted. There are many variants of CRCW PRAM based on the write conflict resolution rule. CRCW model is not used in this thesis.

The other model we use is the *Tree Machine*. *Tree Machine* is a SIMD computer in which the processors are interconnected to form a binary tree. Such a tree has d levels, numbered 0 to $d-1$, and 2^d-1 nodes, each of which is a processor. Figure 2.2 illustrate the architecture for $d=4$. Branches of the tree represent two way links. Each processor at level i is thus connected to a

single parent processor at level $i-1$ and to each of its two child processors at level $i+1$, with the exception of the *root* processor at level 0 (which has no parent) and the *leaf* processors at level $d-1$ (have no children). Through this two-way link, a processor can send or receive a single data item, at a time, to or from, its parent or children. The leaves are the only processors that have an interface with the outside environment and thus handle input and output. All analysis assume that the time taken by a datum to propagate between any two adjacent levels of the tree is constant.

Chapter 3

On Single Row Routing with Prescribed Street Congestion and no Cross-over

3.1 Introduction

As the Single Row Routing Problem is NP-complete [Tsukiyama 82], it is of interest to study useful special cases, which can be solved in reasonable time. In this chapter, we consider the problem, of finding a (not necessarily optimal) layout on a parallel computer, if the maximum street congestion is fixed (maximum number of tracks in upper and lower street are fixed). Han and Sahni in [Han,Sahni 84] have obtained a linear time sequential algorithm for Single-Row Routing with Prescribed Street Congestion. However, that algorithm appears to be inherently sequential.

In this chapter, we consider the case when the prescribed street congestion is 2, i.e. the maximum number of nets possible in upper or lower street is 2. The rest of the chapter is organized as follows. The algorithm of Tsukiyama et. al. [Tsukiyama 80] is briefly described in Section 3.2. The approach adopted for developing an efficient algorithm is described in Section 3.3 while the details in Sections 3.4 and 3.5. The complete algorithm is in Section 3.6.

In Section 3.7, we try to solve the Single-Row Routing Problem when the number of vertical tracks available between consecutive nodes is restricted. The problem first finds brief mentioned towards the end of [Tsukiyama 80]. However, most researchers till Du et. al. [Du 87] assumed that there is always enough room between nodes for allowing wires to cross between streets. The number

of vertical tracks available between any two consecutive nodes will be refered to as the *Cross-over Bound*.

In Section 3.10, we try to augment the parallel implementation of Tsukiyama et. al. algorithm with no *Cross-over* for the case when the upper street congestion is limited to 2 and the lower street congestion is limited to 1. In Section 3.8 and 3.9, we describes the details of parallelization of the Tsukiyama et. al. algorithm. In Section 3.11, we consider the case when the upper and the lower street congestion is restricted to 1. In Section 3.12, we describe how to get ordering, without using topological sort algorithm. Some conclusions are offered in Section 3.13.

For simplicity of presentation, the algorithms as described appear to take $O(\log n)$ time with n processors. Reducing the processors to $\frac{n}{\log n}$ is straight forward and is ommited— basically, we divide the input into $\frac{n}{\log n}$ groups , each containing $\log n$ items, assign a processor to each group, solve the problem for each group sequentially in $O(\log n)$ time , and then proceed as before.

3.2 The Sequential Algorithm

An interval $I = [v_i, v_j]$ in which $c(v_k) \geq 3$ for all $v_k \in I$, and $c(v_{i-1}) = c(v_{j+1}) = 2$, is called a 3-*Interval*. Here, $c(v)$ is the cut-number of node v.

If $N = \{v_{i_1}, \cdots, v_{i_k}\}$ is a net, and $I = [v_i, v_j]$ is an interval then net N *touches* interval I if for some $1 \leq j \leq k$, $v_i \leq v_{i_j} \leq v_j$.

For any interval $I = [v_i, v_j]$ on the Single-Row, let $\overline{L}(I)$ denote set of nets $\{N_k\}$ such that

a) N_k does not touch interval I

b) there are nodes v_a and v_b in N_k such that $v_a < v_i$ and $v_j < v_b$.

$L(I)$ will contain nets which either

a) touch interval I or

b) nets which are in $\overline{L}(I)$.

Example: Consider the net list, L = { N1, N2, N3, N4, N5, N6, N7, N8, N9. N10 }, where, $N1 = (1,11)$, $N2 = (2,7)$, $N3 = (3,5)$, $N4 = (4,6)$, $N8 = (13,17,22)$, $N9 = (14,16)$ and $N10 = (18,20)$ (see Figure 3.1). Let $I^1 = \{4,5\}$. $\overline{L}(I^1) = \{N1, N2\}$ and $L(I^1) = \{N1, N2, N3, N4\}$.

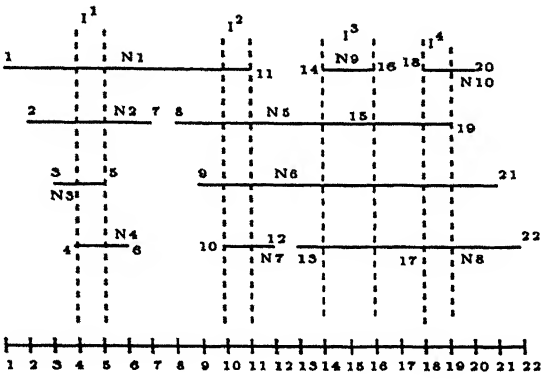


Figure 3.1: Net list for example

Tsukiyama et. al. [Tsukiyama 80] have obtained following necessary and sufficient conditions, if the Street Congestions are to be less than or equal to 2. These conditions are as follows:

1. The cut number of each net should be at most 3.
2. There are at least two nets of cut number at most 2.
3. For every 3-intervals I, there are exactly two nets in $\overline{L}(I)$, i.e., $\mid \overline{L}(I) \mid = 2$.

If these conditions are satisfied, then any ordering of the nets in which, for each 3-Interval I, nets of $L(I) - \overline{L}(I)$ lie in between $L(I)$ will be a valid order. Thus, if nets are laid in successive tracks according to the sequence (ordering), then a valid Interval Graphical Representation will be obtained.

In the above example, $\overline{L}(I^1) = \{N1, N2\}$, $L(I^1) = \{N1, N2, N3, N4\}$, $\overline{L}(I^2) = \{N5, N6\}$, $L(I^2) = \{N1, N7, N5, N6\}$, $\overline{L}(I^3) = \{N6, N8\}$, $L(I^3) = \{N6, N8, N5, N9\}$, $\overline{L}(I^4) = \{N6, N8\}$ and $L(I^4) = \{N6, N8, N5, N10\}$.

The sequence $N8, N10, N9, N5, N7, N1, N6, N3, N4, N2$ is one such order. Figure 3.2 shows the *Interval Graphical Representation* of this sequence.

The algorithm due to Tsukiyama et. al.[Tsukiyama 80] for Single-Row Routing with Prescribed Street Congestion is based on building a directed acyclic graph from sub-graphs where each sub-graph satisfies certain conditions. These conditions are necessary and sufficient, in other words if the conditions for each subgraphs are preserved in the combined graph, then it is possible to obtain a valid sequence by topological sorting of vertices of the graph.

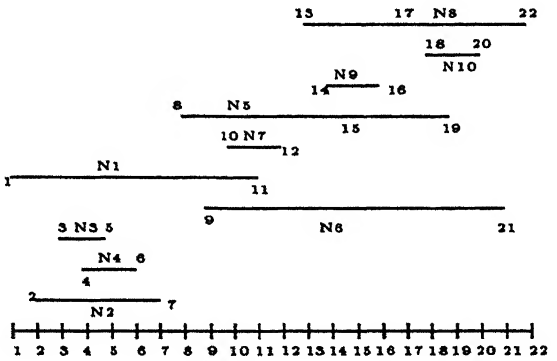


Figure 3.2: Interval Graphical Representation for example

The time complexity of the sequential algorithm (as described in [Tsukiyama 80]) is $O(n^2)$. We will describe a parallel algorithm which takes $O(\log n)$ time with $\frac{n}{\log n}$ processors on a CREW PRAM to get a linked list of ordering. An ordering can be obtained on an EREW PRAM in $O(\log n)$ time with $\frac{n}{\log n}$ processors by list ranking.

3.3 Better Implementation

We model each 3-interval as a directed sub-graph as shown in Figure 3.3. The two overlapping nets which do not have a node in the 3-interval (outer-nets) as well as the the nets which have a node in the 3-interval (inner-nets) are nodes of a sub-graph. Edges are drawn from one of the outer net to both the inner-nets; and from the two inner nets to the other outer-net. There is a dashed-edge between the two inner-nets, the direction of the dashed edge will be fixed later after pre-processing (see Section 3.5). In the over-all graph, direction of all "solid" edges will either be the same as that in the sub-graph, or direction of all solid edges will be reversed.

We try to achieve an ordering such that ordering between any pair of nodes in a sub-graph is the same as the ordering between these nodes in other sub-graphs.

The ordering (direction in figure) between $N2$ and $N3$ is not known *a priori* and will be obtained after pre-processing. Note that "arrows" in a sub-graph may represents either "above" or "below"; we do not know which; we will know only after building the complete graph, whether all arrows in a particular graph represent "above" or "below".

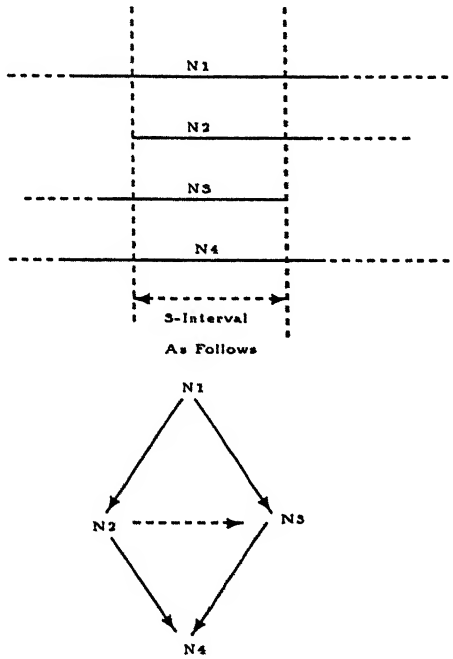


Figure 3.3: Sub-graph representation of each 3-interval

As seen, the partial-ordering of the sub-graph gives us the sequence $N1$, $N2$, $N3$, $N4$ or $N1$, $N3$, $N2$, $N4$. If it is possible to obtain the final graph by reversing the direction of all the edges of the sub-graph, then the partial-ordering of that sub-graph gives us the sequence $N4$, $N3$, $N2$, $N1$ or $N4$, $N2$, $N3$, $N1$. In final ordering one such sequences of each sub-graph will be preserved (we do not know which).

The data-structure we use to represent the sub-graph is as shown in Figure 3.4; we have shown the "assumed" direction between $N2$ and $N3$ as from $N2$ to $N3$ which will be fixed after pre-processing. First row of data structure, stores the information that net N2 is between nets N1 and N4; the second row tells us that N3 is between N1 and N4. Note that only 4-entries in the data structure are "non-trivial". Further processing will be done on this data-structure.

Data structure for sub-graph will be called a *block*. The *blocks* are arranged successively in the order in which they appear along the number-line so that if we take two blocks, *blocks between* them would include all blocks lying between these blocks on the number-line.

Observe that

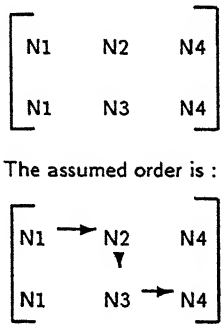


Figure 3.4: Data structure of each sub-graph with assumed direction

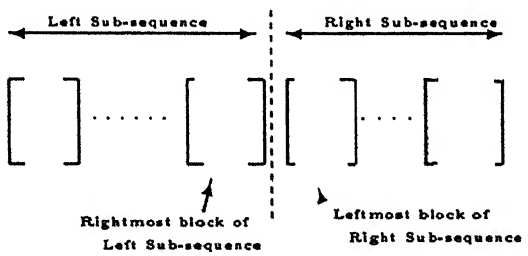


Figure 3.5: The two merging sub-sequences

Observation If net N_i is in two blocks B_1 and B_2 then N_i should also be in all blocks lying between these blocks.

There can be either $0, 1, 2$ or 3 nets which are common to two blocks.

We next describe *reverse* operation, R, on a block as shown in Figure 3.7; the reverse operation reverses the direction of all edges in a block. This can be done in $O(1)$ time sequentially, if bit location is reserved for each block, which can be used to keep track whether the arrows in that block are "direct" or "flipped".

Satisfying the *edge relationship* between two blocks means that if there exists atleast two nets in common between the two blocks then the direction of the edges between them in the two blocks should be the same. The processing is done on the blocks to satisfy the *edge relationship* between any two blocks by merging them.

The processing is done on the blocks in a manner similar to merge sort as shown in Figure 3.6 . While merging two sub-sequences, we only have to concentrate on the leftmost block of the right sub-sequence and the rightmost block of the left sub-sequence (see Theorem 1, below).

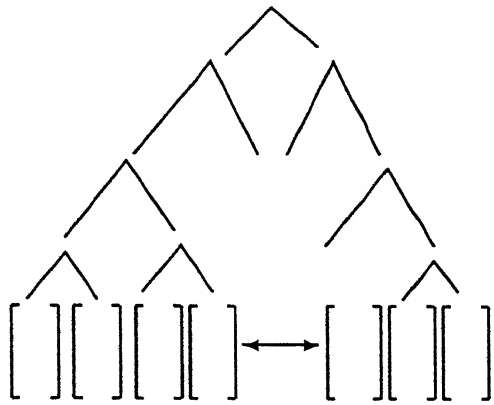


Figure 3.6: The blocks constituting the sub-sequences in various levels of merging

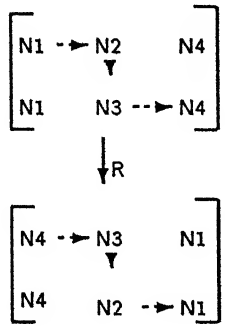


Figure 3.7: The Reverse Operation, R

We call the leftmost block of the right sub-sequence and the rightmost block of the left sub-sequence *merging blocks.*

The relation between two nets must be "consistent" in all blocks in which these nets lie. This is done by using the reverse operation, R, in one block. If reverse operation is applied on one of the two merging blocks, then we also apply the operation on all blocks of the sub-sequence to which that merging block belongs.

As no cycle is created during "merging" of two blocks, the resulting graph will be acyclic (see Theorem 2, below). Hence we can use topological sort to obtain the ordering, and hence the Interval Graphical Representation– an alternate implementation will be described in Section 3.12.

Theorem 1 *If we are able to satisfy the edge relationship between the two merging blocks at time of merging then there cannot be any edge between a pair of*

nets in the left sub-sequence which is of opposite direction (reverse) of another edge between same pair of nets in the right sub-sequence.

Proof: Let us assume that there is such a pair of nets. Then the pair of net must also lie in the merging blocks (by observation before this theorem).

At the time of merging, the edge relationships between the leftmost block of the right sub-sequence and the rest of the right sub-sequence are maintained (either all, or none of the block of a sub-sequence are reversed). Similarly, the edge relationships is maintained for the rightmost block of the left sub-sequence and the rest of the left sub-sequence.

Thus, if there is a violation, i.e. there exists any edge between a pair of nets in the left sub-sequence which is of opposite direction (reverse) of another edge between same pair of nets in the right sub-sequence, then there must be a violation in the *merging blocks* also (see observation before the theorem). But "merging" the two sub-sequences ensures that edge relationship between them has been satisfied. Hence it is impossible to have such violation. ∎

Theorem 2 *The graph obtained by merging the two sub-graphs is acyclic.*

Proof: Consider the merging step till which all sub-graphs were acyclic. Let us assume that merging of two sub-graphs now result in a cycle. For a set of nets satisfying the Tsukiyama et.al. conditions, we model each 3-Interval as a sub-graph as shown in Figure 3.3. Here $N2$ and $N3$ are between $N1$ and $N4$ in the partial ordering of the sub-graph. The direction between $N2$ and $N3$ is not known with respect to the rest of the graph as shown in the figure with a dashed edge.

However, after preprocessing as described in section 3.5, the direction of the dashed edge will be fixed with respect to the rest of the sub-graph; thus the dashed edge will be converted to a solid edge such that its direction will be changed only when the direction of the rest of the edges in the sub-graph are changed.

We, therefore, will be working with sub-graphs as shown in Figure 3.8. There is no edge between $N1$ and $N4$ but it can inferred from the sub-graph.

Going back to the proof of the theorem, we have two sub-graphs which are themselves acyclic. There can be four cases depending on the number of nets common between them.

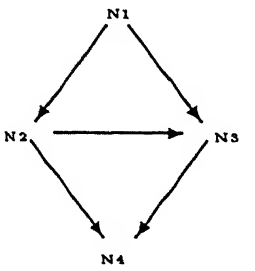


Figure 3.8: The sub-graph after the direction of dashed edge is fixed

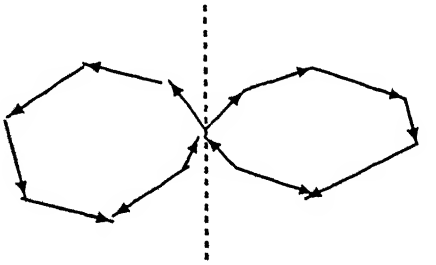


Figure 3.9: Two sub-graphs with one node common

Case1 [No common net] There can be no cycle in this case because we have two disjoint sub-graphs which are themselves acyclic.

Case 2 [One net common] As shown in Figure 3.9, we have two acyclic sub-graphs having one node common. The cycle in the resulting graph must have formed with edges from both the sub-graphs. But then, the cycle must have paseed through the common node twice. So, it is not possible to have a cycle in this case.

Case 3 [Two common nets] In case there exist an edge between the common nodes (A and B in Figure 3.10) then formation of a cycle in the resulting

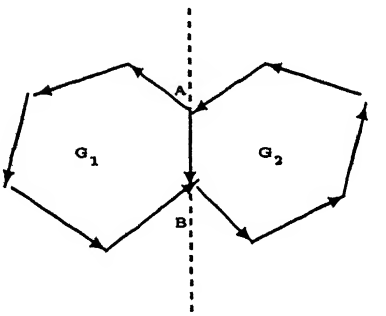


Figure 3.10: Two sub-graphs with two nodes common

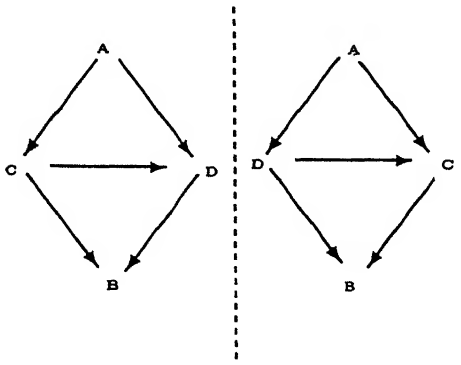


Figure 3.11:

graph will imply that there already exist a cycle in one of the sub-graph (G_2 here).

Since our initial sub-graphs are acyclic, it means that the sub-graph in which a cycle has been found is otherwise acyclic if the edge between the common node is not there. This edge was contributed by the other sub-graph (G_1 here) only.

In the merging blocks to which A and B belong, there cannot be an edge between A and B only if they are the outer-nets (A and B are $N1$ and $N4$ or vice-versa, see Figure 3.3). Since it has been inferred at the time of merging that direction of the edge between A and B with respect to the rest of the sub-graph is from A to B, the only possible sub-graphs represented by one of the merging block (contributed by G_2) are the ones shown in Figure 3.11.

If we consider the sub-graph G_2 with specific edge-relationship between A and B obtained from Figure 3.11, then either of the graphs shown in Figure 3.12 is possible. We see that both of them have a cycle. Hence, the sub-graph G_2 must have had a cycle before merging has been done. A contradiction.

Case 4 [3 common nets] In the sub-graph represented by either merging blocks, any 3 nodes will atleast have the edges shown in Figure 3.13, where $N1, N2$ and $N3$ are the common nodes. Edge $(N1, N3)$ will not exist if $N1$ and $N3$ are outer-nets. In Figure 3.13, $N2$ is an inner-net and either $N1$ or $N3$ or both $N1$ and $N3$ are outer-nets.

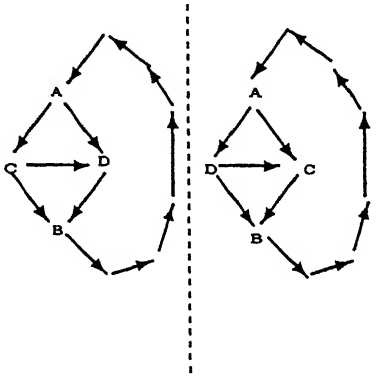


Figure 3.12: The two graphs with the graphs in the previous figure as sub-graph

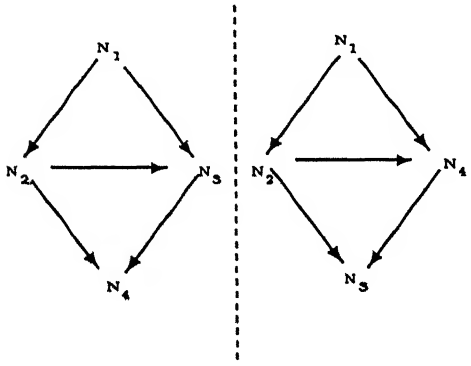


Figure 3.13:

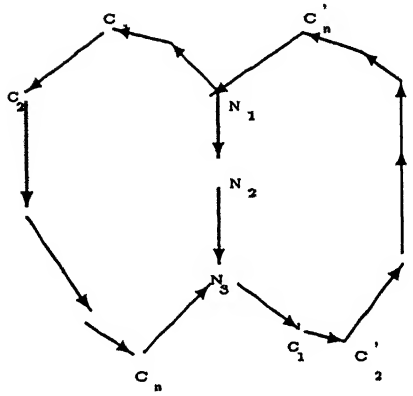


Figure 3.14: The graph with the graphs in the previous figure as sub-graphs

Considering sub-graphs G_1 and G_2 having either of the sub-graphs shown in Figure 3.13 common to both of them as well as a cycle (C_1, C_2, ..., C_n, N_1, C'_1, C'_2, ..., C'_n, N_3) in the resulting graph, we can infer that one of them must have a cycle before merging as shown in Figure 3.14 (G_2 here). A contradiction.

Hence the resulting graph is acyclic.

∎

3.4 Merging of Blocks

At the time of merging of blocks, four cases may arise depending on the number of nets common between them. These are:

Case 1 There is no net common between them. We do nothing in this case.

Case 2 There is atmost one net in common to the two merging blocks. In this case, we do nothing.

Case 3 There are two nets common to the two merging blocks. In case of outer-nets, there is no direct edge-relationship. They have to be inferred from the sub-graph. Here, either same edge relationship exists in both blocks or the relationship in one is the reverse of that in the other. In the first case, no operation is done and in the second case the reverse operation is applied to one of the sub-sequence.

Case 4 When there are three nets common to the two merging blocks. This case is considered in the next section.

3.5 Pre-processing

Till now, we have assumed that at the time of merging, direction of dashed edge is fixed with respect to the rest of the sub-graph in each block. Further, its reversal will mean reversing the direction of all remaining edges of the sub-graph. In this section, we will show how to fix this direction by "pre-processing" of blocks.

We try to obtain the direction of the dashed edge as early as possible. In case a block has 3 nets in common with a neighboring block, its direction can

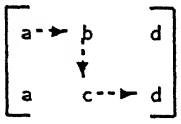


Figure 3.15:

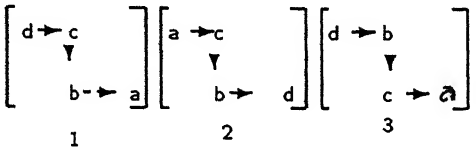


Figure 3.16: The three blocks that can be merged with the block in the previous figure neglecting dashed edge

be fixed for certain as described later. In case the direction cannot be fixed for certain after pre-processing, we can assign any direction to the dashed edge.

We, in fact, start with an assumed direction for each dashed edge. If the assumed direction is not changed after pre-processing, then the assumed direction is the final direction.

In case a block has 2 nets in common with one of its neighboring blocks, then the direction of dashed edge is not fixed, as either direction will do (reversal of one block during *merging* will be sufficient). The direction will be fixed only if the other neighboring block has 3 nets in common with it. Similar is the case for 0 and 1 common nets case.

From the previous section, it is clear that the reverse operation is sufficient for merger between *blocks* in case there are 0, 1 or 2 common nets between them. But in the 3-common nets case, simply reversing the direction of the edges in one of the two sub-graph may not be sufficient. Consider the block shown in Figure 3.15, three configurations are possible, all satisfying the Tsukiyama et. al. conditions (b and c are between a and d), as shown in Figure 3.16. It is possible to merge the block in Figure 3.15 with all the blocks of Figure 3.16 provided the dashed-edge between the inner-nets in the second and third block was in the reversed direction. From these we can infer that assumed initial direction of the edge between the inner-nets in each blocks is crucial.

In pre-processing, we will concentrate on sub-sequence in which each consecutive pair of blocks have three nets in common (see Figure 3.17). The line in each block is formed by joining the elements which are in common to the pair of blocks currently under consideration; if a common net occurs twice in a block, only one occurrence is selected. For example, the block in Figure 3.15

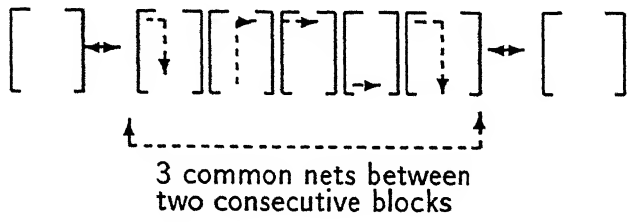


Figure 3.17: Each blocks in sub-sequence has 3 nets in common with its neighboring block(s) in sub-sequence

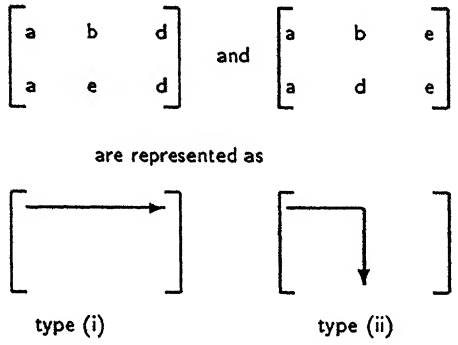


Figure 3.18: The blocks in common with the block of Figure 3.15 and their representation

has nets a, b and d in common with the blocks shown in Figure 3.18, the lines are shown in lower half of Figure 3.18. The block in Figure 3.15 is referred to as the *representative block*.

At first, we try to find the types of block which can have three common nets with the *representative block*. Without loss of generality, we can assume that the other blocks (with which the *representative block* shown in Figure 3.15 has three nets in common) are the ones shown in Figure 3.18; these blocks are marked as type(i) and type(ii) blocks.

Type(i) block are such that the direction of dashed edge (between b and e) cannot be determined as all the common elements occupy the same position in both blocks.

However, in type(ii) blocks, the direction of the dashed edge can be determined.

We, first find a maximal continuous sub-sequences of type(i) blocks. The block on the left of the leftmost block and the block on the right of the rightmost block of such sub-sequence is either a type(ii) block or is a block having fewer than 3 nets common with it.

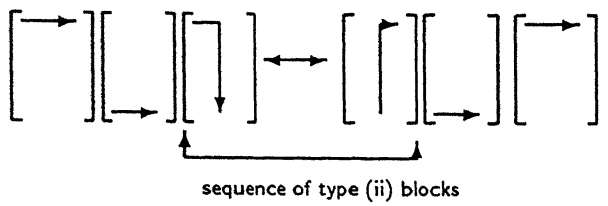


Figure 3.19: Sub-sequence of successive type(ii) blocks

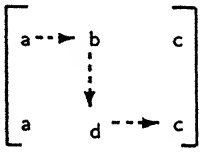


Figure 3.20: Dual Representative Block

Type(ii) block adjacent to a type(i) block, gets the direction of dashed edge fixed. This direction is obtained by looking at the neighbouring type(i) block. In remaining cases, merging can be done directly. We, thus, have an arrangement as shown in Figure 3.19

We thus, look at a maximal sequence of blocks having 3-common nets, and further, in each block, the inner-nets are the common nets. These blocks are *type(ii)* blocks. Without loss of generality, assume that each type(ii) block is as shown in the Figure 3.20. This will be called the *dual representative block of the representative block* shown in Figure 3.15, or for short, *dual representative block*. There are 24 possible blocks having a, b, d and e as non-trivial elements. Out of these in only twelve cases, it will be possible to get a "valid" ordering; these are shown in Figure 3.21. In remaining 12 cases, no ordering is possible. These are shown in Figure 3.22.

Of the twelve blocks shown in Figure 3.21, blocks numbered 1, 2, 11 and 12 are type (i) blocks (with respect to dual representative block). Note that the dual representative block is of type(ii); hence its direction is fixed in a manner already described. Only merger with blocks numbered 3, 4, 5, 6, 7, 8, 9 and 10 of Figure 3.21 remains to be considered.

The blocks numbered 6 and 9 are inter-related in the sense that one is the reverse of the other. For each *dual representative block*, we find out whether one of it's neighbouring block, i.e. the block before or after it in the number-line, is of type 6 or 9. If this is so, then the direction of the dashed-edge of the *dual representative block* remains as it is whereas that of the neighbouring block changes. This is because if we try to merge the various combinations

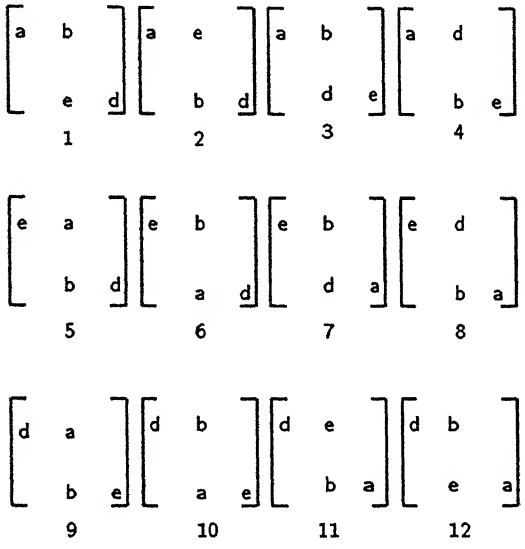


Figure 3.21: Cases where valid ordering is possible

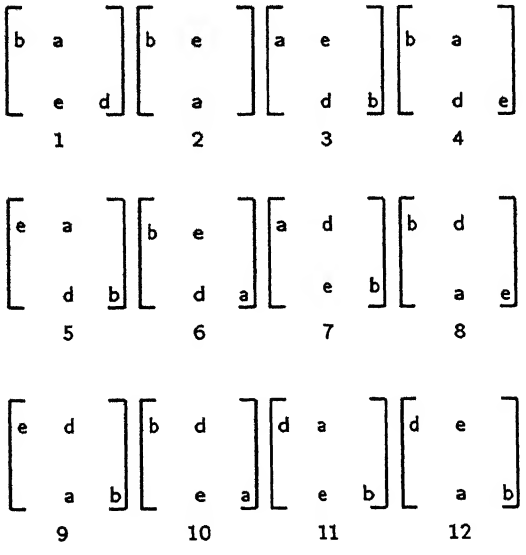


Figure 3.22: Cases where no valid ordering is possible

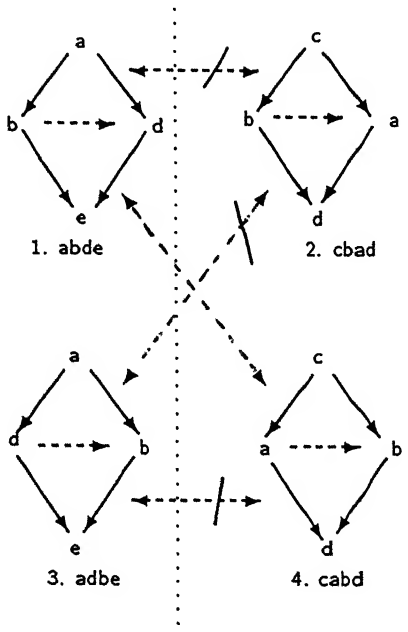


Figure 3.23:

possible from the two sub-graphs represented by the *dual representative block* and the neighbouring block, neglecting the direction of the dashed-edge (see Figure 3.23), then only one combination is possible.

The case for blocks numbered 5 and 10 is similar. Here also, only one combination of the direction of the dashed edge (between the inner-nets) is possible. In this case, the assumed direction of the dashed edge remains as it, (i.e., "fixed").

Coming to blocks numbered 3, 4, 7 and 8 of Figure 3.21, dashed edge is between b and d. Considering the graph for each such blocks, we have a sequence of graphs such as the one shown in Figure 3.24. However, in this case (see Figure 3.25), the direction of the dashed edge of the dual representative block cannot be fixed, as two combinations are valid.

We merge all blocks continuous sub-sequences of type(ii) blocks— i.e., blocks of type 3, 4, 7 and 8, having same elements as inner-nets. This is done prior to the final merging in a manner similar to merge sort. While merging, it is quite possible, that direction of the dashed edge in one of the sub-sequence may have to be changed. Then the direction of the dashed edges in all the blocks of that sub-sequence can be changed without destroying the acyclic property because we are, in fact, working with a graph such as one shown in

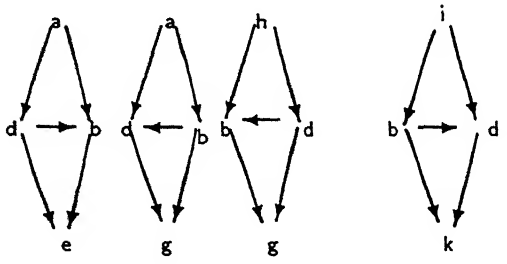


Figure 3.24: The successive type(ii) blocks with the same elements in the dashed edge

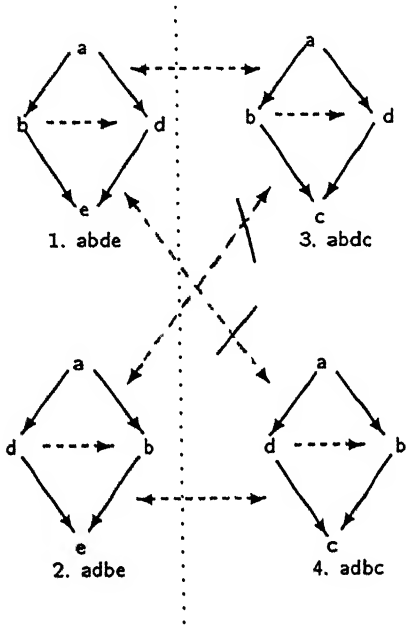


Figure 3.25:

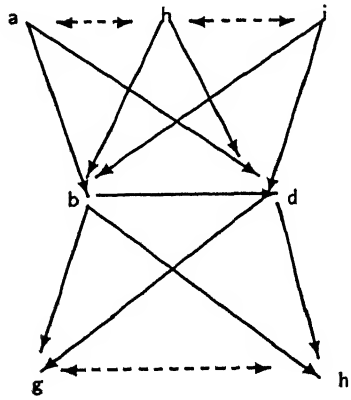


Figure 3.26: The graph with b and d as the inner nets

Figure 3.26 where the direction of the edge (b, d) is immaterial. This can be done in $O(\log n)$ time as the height of the tree is $O(\log n)$. The final direction of the edge (b, d) can be determined exactly if the neighboring block of such a continuous sub-sequence is of type other than type 3, 4, 7 and 8. Otherwise any direction of edge (b, d) will be valid.

After the above pre-processing has been done, if the direction of any dashed edge is not fixed yet, then the assumed direction can be taken as the final direction with respect to the rest of the block.

If, at any point of time, the direction of any dashed edge whose direction has been fixed needed to be changed with respect to the rest of the block then it is not possible to have an order satisfying the prescribed street congestion as the direction of a dashed edge is fixed only after it can be said for certain that that direction is the required one.

After all the above preprocessing has been done, we are left with blocks such that merging between two neighbouring sub-sequences can be done in constant time. If merging is not possible, an order satisfying the prescribed street congestion does not exist. The time complexity depends on the height of the tree which $O(\log n)$. Hence, the time complexity is $O(\log n)$.

3.6 Parallel Algorithm

Parallel Algorithm is as follows:

Step 1 Identify *type(i)* blocks by comparing each block with it's neighboring block. If the neighbouring block is a *type(ii)* block then assign direction to the dashed edge.

Step 2 Find sub-sequences of type(ii) blocks which have the same elements in the dashed edge position (see Figure 3.24). Merge them as described in Section 3.5.

Step 3 For each block which is of type(ii) (with respect to its neighboring block) and whose neighboring block is of type other than 3, 4, 7 and 8, the direction of the dashed edge is fixed for certain as described in Section 3.5..

Step 4 Two successive blocks can now be merged. Merge them in a manner similar to merge sort with one processor assigned to each block. Merging

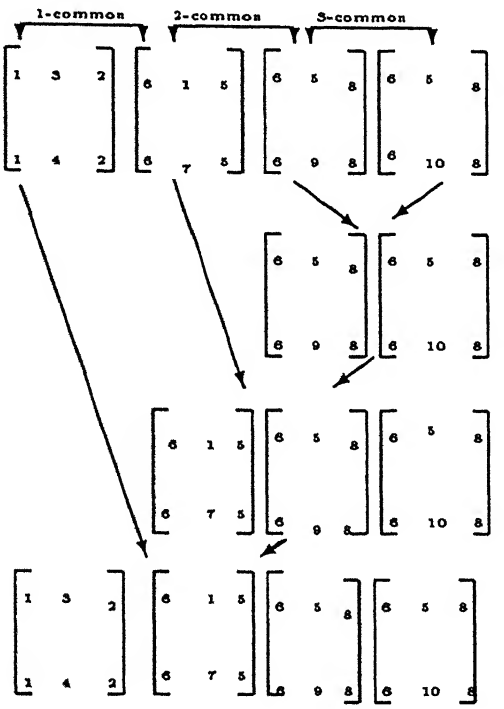


Figure 3.27: The Parallel Algorithm tried on the example of Figure 3.2

of two sub-sequence here means obtaining the same edge relationships of each edge if it exists in more than one block. During merging, only the *merging blocks* are concentrated upon— if the "reverse" operation is applied to one of them, the same operation is applied to all the blocks of the sub-sequence to which that merging block belongs to.

Note that if a valid ordering exists, then it is always possible to merge the blocks. Thus, if at any point of time the merging process "fails" then we can straight away stop the algorithm, as in this case, a valid ordering cannot be generated.

Clearly, the algorithm takes $O(\log n)$ time with n processors. For the example of Section 3.3, the algorithm, is illustrated in Figure 3.27. Topological sort on the graph formed by merging these subgraph gives following ordering: $N6$, $N7$, $N1$, $N3$, $N4$, $N2$, $N5$, $N10$, $N9$, $N8$. The interval graphical representation for the example is shown in Figure 3.28.

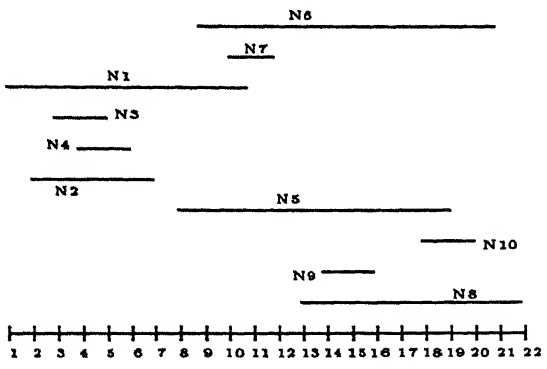


Figure 3.28: The Interval Graphical Representation of the nets laid down in successive tracks according to the obtained sequence

3.7 Modification for no Cross-over Case

In this section, we try to solve the *Single-Row Routing* Problem when the number of vertical tracks available between consecutive nodes is restricted. The problem first finds brief mentioned towards the end of [Tsukiyama 80]. However, most researchers till Du et. al. [Du 87] assumed that there is always enough room between nodes for allowing wires to cross between streets. The number of vertical tracks available between any two consecutive nodes will be refered to as the *Cross-over Bound*.

We are required to minimize the total area, A, of the layout; A can be calculated as follows

$$A = (K + 1) * (n - 1) * (M_u + M_l)$$

here n is the number of nodes, K is the maximum number of Cross-overs between any two adjacent nodes in the layout and M_u and M_l are the maximum number of tracks used in upper and lower streets, respectively.

Han and Sahni [Han,Sahni 84], proposed a linear time algorithm for the case when the number of tracks available in each street is restricted to a small constant (without considering cross-over constraints).

Using this, Du et. al. [Du 87] obtained a linear time algorithm to solve the Single-Row routing problem with Cross-over Bound and Prescribed Street Congestion. They basically considers all possible permutation that can "arrive" at nodes which are starting points for the given nets. However, the algorithm of Du et. al. [Du 87], appears to be inherently sequential.

In this chapter, we use some results of Du et. al. [Du 87] together with algorithm of Tsukiyama et al.[Tsukiyama 80] to obtain a parallel algorithm for Single-Row Routing with no *Cross-over* (K =0) and a prescribed street congestion of 2 in upper-street and 1 in lower street. Earlier Saxena and Prasad [Saxena 89] have obtained an $O(\log n \log\log n)$ time parallel algorithm with n processors for the problem of finding a layout without inter-street crossings. However, they did not consider the case when street congestion is prescribed. Moreover, the layout obtained by their algorithm is in general not optimal.

3.8 The Tsukiyama et. al. algorithm for Single-Row Routing with K_u=2 and K_l=1

Let K_u and K_l denote the prescribed upper and lower street congestions—maximum number of nets possible above and below the single row. Let $c(v)$ denote the cut-number of node v.

The algorithm of Tsukiyama et. al. [Tsukiyama 80] for $K_u = 2$ and $K_l = 1$, is similar to the algorithm described in section 3.6. It involves obtaining a permutation of nets, such that when nets are laid out in successive tracks, we obtain the Interval Graphical Representation.

An interval $I = [v_i, v_j]$ will be referred to as *2-interval* if $c(v_k) \geq 2$ for all v_k on I and $c(v_{i-1}) = c(v_{j+1}) = 1$.

The algorithm of Tsukiyama et. al. is [Tsukiyama 80]:

Step 1 Identify all 2-intervals. This can be done in $O(\log n)$ time by combining two sub-sequences of 2-intervals having a common node, at a time, in a manner similar to merge-sort; the initial "sub-sequences" (for first merge) will consist be all intervals $I = [v_i, v_{i+1}]$ which have a node whose *cut-number* is greater than or equal to 2.

Step 2 If there is no 2-interval and $q_m \leq 2$, any permutation of the nets will do; goto Step 4.

Step 3 If the conditions in Step 2 fails, then any permutation for which the following condition is satisfied will do:

> For each 2-interval I, all nets N, which overlaps I but have no node in I, preceeds all the other nets which overlap I but have a node in I.

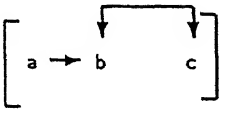


Figure 3.29: A block for $K_u = 2$ and $K_l = 1$ case

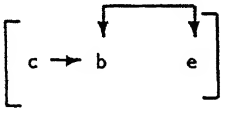


Figure 3.30:

Step 4 If the nets are laid out in successive tracks in the order given by the permutation obtained in either Step 2 or Step 3, the Interval Graphical representation will be obtained.

3.9 Parallelization of Tsukiyama et. al. algorithm

Note that no preprocessing needed to be done this time because it is always possible to merge two successive blocks if any permutation is feasible.

The *blocks* are of type shown in Figure 3.29. Each block describes one 2-Interval as obtained from Step 3 in the algorithm described in Section 3.8; a is the net which over-laps with the 2-Interval but has no node in it; b and c overlap with the 2-Interval besides having a node in it.

The position of a is fixed; however the positions of b and c can be interchanged during *merging*. In this case, there is no need for any *reverse operation* because the direction of the edge (such as the one between a and b in Figure 3.29) is fixed. The direction of the edge between b and c has to be taken care of.

The direction of the edge between b and c is obtained when we are merging it with the *block* shown in the Figure 3.30; the resulting *block* is shown in Figure 3.31. Each block can be of either of the two types shown in Figure 3.32. In first block, the positions of b and c can be interchanged whereas in the second block, the positions of b and c are fixed.

Note that two blocks may either have zero, one or two nets in common. If they have zero or one net in common, we leave them as they are. If they have two nets in common, say a and b, then a and b may occupy either of the

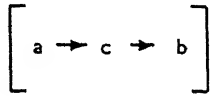


Figure 3.31:

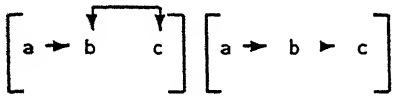


Figure 3.32:

position shown in the Figure 3.33. Let us discuss how these blocks are merged with the block shown in Figure 3.34,

Merger of the first or the second block of Figure 3.33 with the block shown in Figure 3.34 results in no change in either of the blocks.

Only the merger of the third block with the block shown in Figure 3.34 is non-trivial because the direction of the edge of the third block gets fixed as shown in Figure 3.35; here we have assumed that the third block of Figure 3.33 belongs to the right sub-sequence and the block shown in Figure 3.34 belongs to the left sub-sequence.

For any other block in the right subsequence, if the relation between a and e is yet to be decided then the relationship between a and e in the merging blocks is conveyed to them. This can be done in $O(1)$ time because each block in the right sub-sequence can collect this information from the merging block independently.

Example : Let $N1 = \{1,5\}$, $N2 = \{2,4\}$, $N3 = \{3,8\}$, $N4 = \{7,10\}$, $N5 = \{6,14\}$, $N6 = \{9,11,13\}$ and $N7 = \{12,15\}$. Then, $N1, N2, N5, N6, N7, N3, N4$

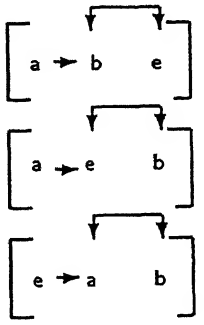


Figure 3.33:

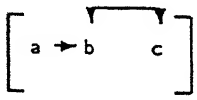


Figure 3.34:

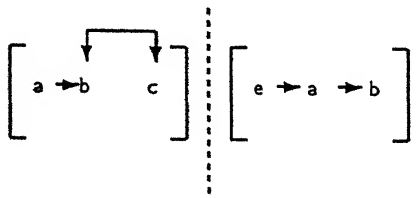


Figure 3.35:

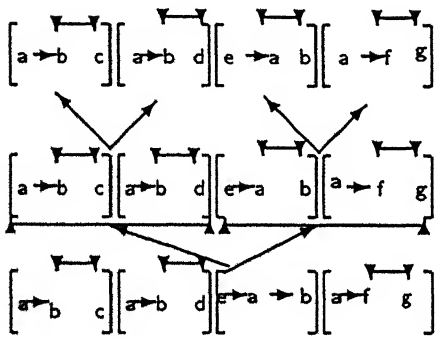


Figure 3.36: The way merging is done between blocks

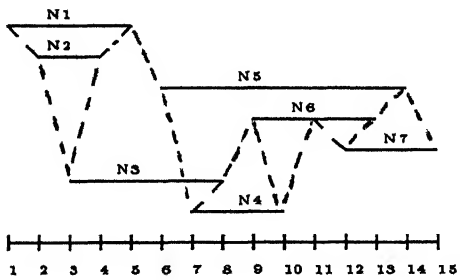


Figure 3.37: The Interval Graphical Representation of the example net list

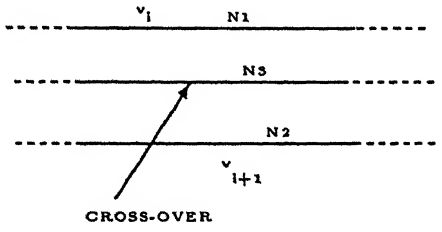


Figure 3.38: Cross-over between two successive nodes

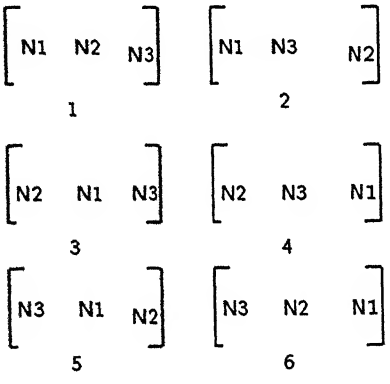


Figure 3.39:

is a desired permutation. The Interval Graphical Representation is shown in Figure 3.37. Here, there is one Cross-over— between nets 6 and 7.

3.10 Without inter-street Cross-overs

In Interval Graphical Representation, a *cross-over* occurs whenever reference line cuts a net between two nodes. In Figure 3.38, $N3$ is cut by the reference line between v_i and v_{i+1}. For this example, we can list all possible permutation of $N1$, $N2$ and $N3$ as shown in the Figure 3.39. The permutation numbered 2 and 4 will result in inter-street crossings, and cannot be allowed. In remaining four permutations, there is no Cross-over as $N3$ is not between $N1$ and $N2$.

Thus, it is quite possible that a permutation such as the one shown in the Figure 3.40 , could get converted into one of the type shown in Figure 3.41 and

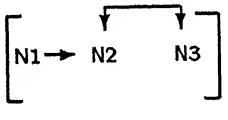


Figure 3.40:

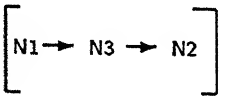


Figure 3.41:

should not be allowed as it will result in *Cross-over.*

On Concurrent Read Exclusive Read (CREW) model, if a processor is assigned to each block, then that processor can in $O(1)$ time, modify the block (if necessary) to insure that there will be no cross-over. The modified blocks are merged as before. Clearly, the complexity of the algorithm remains un-changed.

For the previous example, we do the modifications on the blocks as shown in the Figure 3.42

After merging, we get a permutation like— $N1, N5, N2, N4, N6, N7, N3$. The corresponding Interval Graphical Representation is shown in Figure 3.43. Observe that there are no Cross-overs.

3.11 The $K_u = 1$ and $K_l = 1$ case

Theorem 3 *In the case of $K_u = 1$ and $K_l = 1$, if a realization is possible with Cross-overs, then a realization without Cross-over is also possible.*

Proof: In case $K_u = 1$ and $K_l = 1$, we should note that at any node of the Single-Row, atmost 2 nets are covering it. We divide the *number-line* into *intervals* of successive nodes such that in two intervals, atleast one net is different. We consider one interval at a time (from left) in the order in which they occur on the number-line.

In Figures 3.44 and 3.45, we have I_1 as the first interval. Without loss of generality, we assume that $N2$ ends before $N1$. I_1 can be made to have no cross-over by ending the interval just before the cross-over. It is not possible for $N2$ to have a cross-over because, as shown in Figure 3.46, this would mean that the upper-street allows 2 nets in case $N2$ lies below the Single-Row.

Depending on whether net $N3$ ends before $N1$ or not, we will have two different cases as shown in Figures 3.44 and 3.45.

It is possible to pass over the cross-over to the next interval (I_3 here) in both the cases as shown in Figures 3.47 and 3.48. Hence, I_2 does not have a cross-over now.

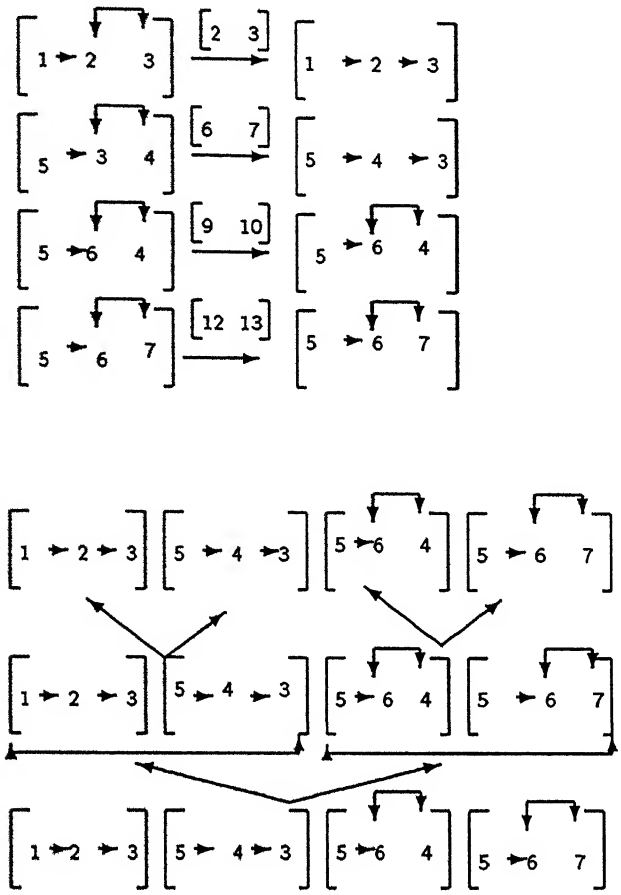


Figure 3.42: The Algorithm runned on the example

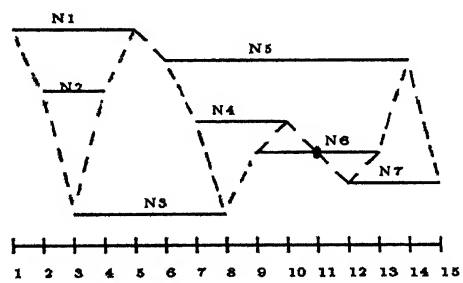


Figure 3.43: The Interval Graphical Representation with no Cross-over

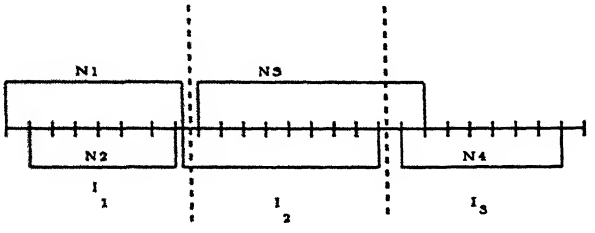


Figure 3.44:

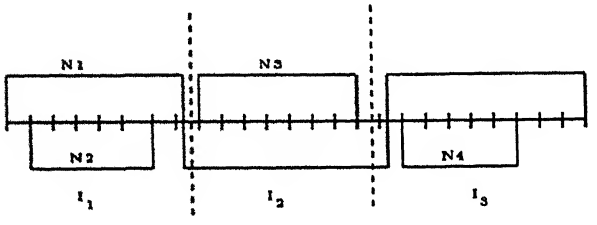


Figure 3.45:

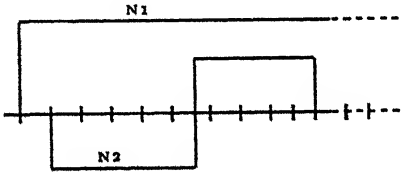


Figure 3.46:

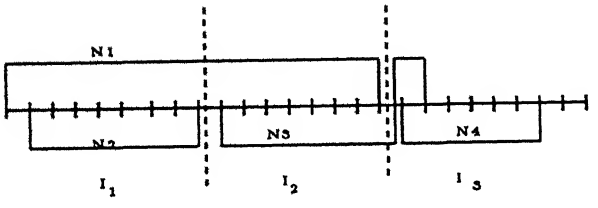


Figure 3.47:

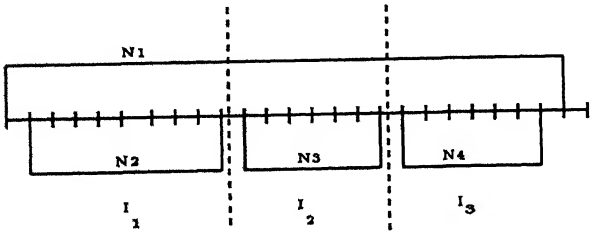


Figure 3.48:

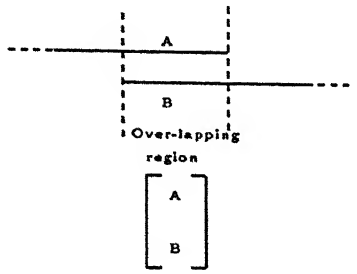


Figure 3.49:

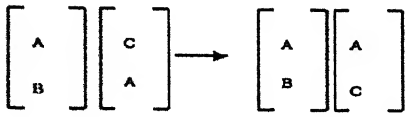


Figure 3.50:

Likewise, considering the next intervals I_3 and the other intervals in turn, we can remove cross-overs from each interval.

Thus, it is possible to have a realization for $K_u = 1$ and $K_l = 1$ without cross-over iff a realization is possible with a cross-over. ∎

From the above theorem, it is clear that whenever a realization is possible for $K_u = 1$ and $K_l = 1$, there is another realization without cross-overs. As we can have atmost two nets overlapping any node. If we are able to find out the net which will lie above and below the node axis, we can get a realization by just placing the net which lie above the node axis in the upper track and making the node connection— joining the net to the nodes by vertical lines. We proceed similarly for nets which lie below the node axis. This realization is without Cross-overs. We only have to find the nets which lie above and below the node axis.

We model each pair of over-lapping nets as a block as shown in Figure 3.49. These blocks are assigned to successive processors according to the order in which the over-lapping pairs of nets represented by them appear on the node axis.

From observation made earlier, if a net exist between two blocks then it must exist in all the blocks that lie between them on the node axis.

We merge these blocks in the manner similar to merge sort described earlier. Here we consider two sub-sequence of successive blocks at a time. At the time of merging we consider the leftmost block of the right sub-sequence and the rightmost block of the left sub-sequence. If there is a net common between

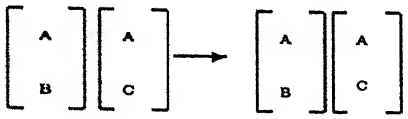
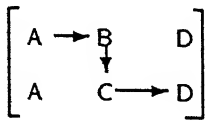


Figure 3.51:



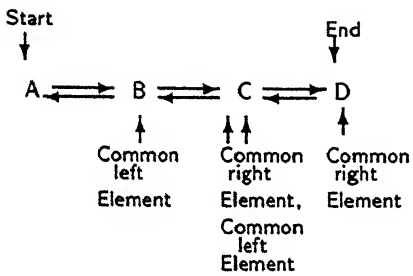


Figure 3.52:

them then we try to ensure that both of them occupy the same row as shown in Figure 3.50 and Figure 3.51. In case if blocks are as in Figure 3.50 then it is necessary that all other blocks in the sub-sequence (to which the merging block belongs) are also reversed. We can assign all nets which are in the top row position to the upper track and the nets in the bottom row position to the lower track. This realization is without Cross-overs.

3.12 Removing Topological Sort Bottleneck

To obtain final ordering, it is necessary to carry out topological sort on the directed acyclic graph created. Unfortunately, topological sort is a costly operation—it takes $O(\log n)$ time with $O(\frac{n^3}{\log n})$ processors. In this section we describe an alternate method. The basic observation is that after merging, direction of all solid and dashed edges are fixed; moreover, no block can be now reversed. Hence, we can now replace each block by linked list, and carry out the merging algorithm once again. In rest of this section we describe, the details.

We start by modelling each block as a doubly link list as shown in Figure 3.52. Here, *common right element* pointer are pointer to the nodes which are in common to some nodes in the doubly link list of the block which lies to

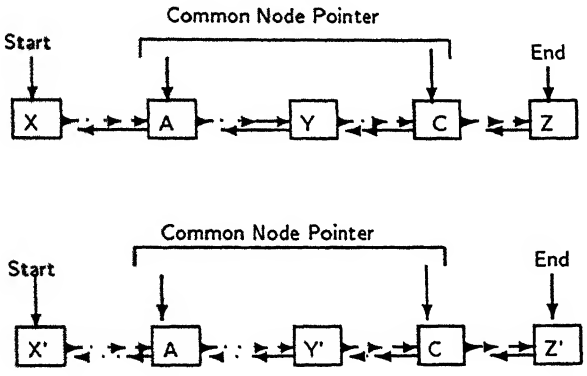


Figure 3.53: The two doubly link lists to be merged

the right of it. *Common left element* is similarly defined. *Start* and *End* pointers are pointers to the start and the end of the link list respectively.

We will modify these link list in same order in which merging of blocks has been done. A link list will represent the sub-sequence of blocks considered at different stages of merging. Here, instead of considering the different blocks to be merged, we consider the link lists and try to combine two link lists representing the two sub-sequence which are being merged.

Since we have kept the pointers to the common elements in the beginning, these remains preserved as address of these nodes do not change; only the *inter-link*ing gets changed while combining the link lists. Thus the common elements between two link lists representing two successive sub-sequence of blocks have already been determined at the time of merging of lists (thus traversal of linked list is not required). We thus have two doubly link lists as shown in Figure 3.53

We merge them as shown in Figure 3.54 Here, $A-$ and $A+$ denotes the element just before and after A, respectively, in the link list. Similarly with $C-$ and $C+$. These pointer can be obtained easily by traversing the specific link list forward or backward one step from either A or C. The *inter-linking* can now be easily achieved as the nodes between which *inter-linking* is required has pointers to them.

The Figure 3.54 describes merging between two link list at one level of merging. There are $O(\log n)$ such levels, so the total time complexity is $O(\log n)$ as each merging can be done in constant time.

The link list, thus obtained, can now be *rank*ed in $O(\log n)$ time. The rank of each node (representing a net) will give us the track in which that net will lie. Hence the *Interval Graphical Representation* can be obtained.

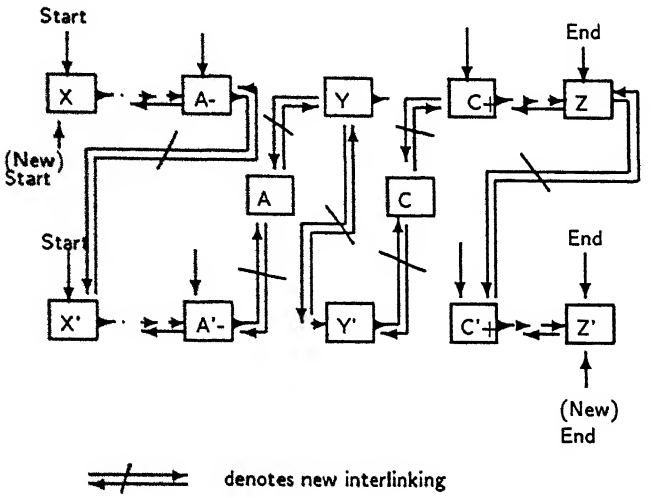


Figure 3.54: The two link lists in the previous figure being merged

3.13 Conclusions

In this chapter, we have considered whether a realization is possible in cases where the upper and the lower street can allow limited number on tracks. Since we were able to find out whether a realization is possible in the following cases,

- Case 1. $K_u = 1$ and $K_l = 1$
- Case 2. $K_u = 2$ and $K_l = 1$
- Case 3. $K_u = 2$ and $K_l = 2$

it is possible to find out optimum realization in case it is possible to find out a realization with Case 3 as the Prescribed Street Congestions. Trying Case 1, Case 2 and Case 3 as the Prescribed Street Congestion in that order and finding out the first case that results in a realization will give us the optimal realization.

Chapter 4

Implementation on Tree Machine

In this chapter, we will discuss an implementation of the algorithm of Section 3.6 for $K_u = K_l = 2$ on a *Tree Machine*.

The algorithm had three different phases:

1. Preprocessing,
2. Merger of blocks and
3. Obtaining the final order.

For the pre-processing part it is necessary for each block to communicate with its two neighbors. While blocks assigned to odd numbered processor can communicate with their right neighbor through the processor in the level just above the leaf-level, every block assigned to even numbered processor cannot communicate directly with its right neighbor (to get the direction of its dashed edge fixed). How-ever, by modification of Akl's algorithm for prefix computation[Akl 89, Meijer,Akl 87], each leaf can find out the value stored in the previous leaf in $O(\log n)$ time (see e.g., [Saxena 94] for details).

Thus each block can get the information about its previous block in $O(\log n)$ time for the preprocessing step.

In Section 4.2, we describe the second step and in Section 4.3, we describe the third step.

4.1 Merger of blocks on Tree Machine

We assign successive *blocks* needed to be merged to successive *processors* in the leaves. Thus, each leaf can now determine in $O(1)$ time, whether it needs to be reversed or not. Value 1 is assigned to each processor if the *block* needs to

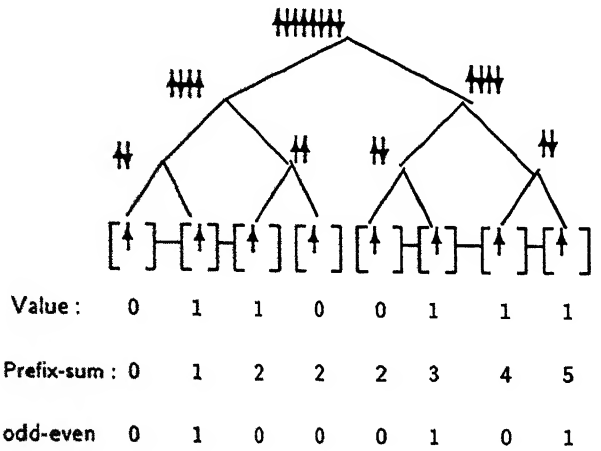


Figure 4.1: Various levels of Tree-machine being merged

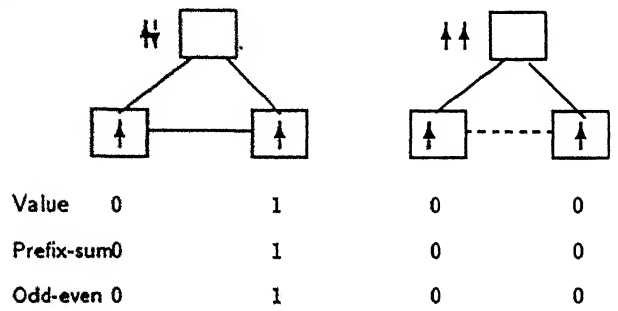


Figure 4.2: Blocks merger for $height = 1$

be *reversed* with respect to the previous *block*. Otherwise value 0 is assigned to the *block*. The very first block is assigned 0. We compute the prefix-sum of the values in $O(\log n)$ time using Akl's algorithm on prefix-sum on trees [Akl 89, Meijer,Akl 87]. The odd-even value of the prefix-sum of each *block* will tell us whether a block needed to be reversed or not. In case of odd value, it need to be reversed; otherwise not (see theorem below). Since each block has four items, it can be reversed in $O(1)$ time.

Theorem 4 *The odd-even value of the prefix-sum of each block will denote whether a block need to be reversed or not; odd value denoting reversal.*

Proof: Figure 4.1 shows the merged sequence at various levels of the tree. Two leaf nodes are connected by a solid edge if one of them need to be reversed with respect to the other. Here, the blocks in the right sub-sequence are reversed in case a reversal is required. The direction of the arrow at the leaf-level denote initial direction. Direction at the root-level denotes the final direction.

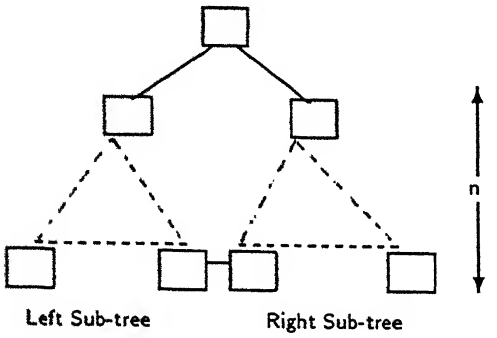


Figure 4.3: Blocks merger for $height = n + 1$

We prove the Theorem by induction on height of tree. For height=1 the theorem is clearly true. (see Figure 4.2).

Let us assume that the theorem is true for height $= n$. To prove that the theorem is also true for height $= n+1$, let us consider a node at height $= n+1$ (See Figure 4.3). We have two sub-tree at this node of height n. By induction hypothesis, the theorem is valid for both these sub-trees. For the right sub-tree, the values of all blocks can be changed from odd to even or even to odd if a block is added in front ot these blocks with a value 1. Otherwise they remain as it is.

In case the left-most block of the right sub-tree is connected by a solid edge with the right-most block of the left sub-tree, (the prefix-sum will differ by 1) whatever may be the value of the left-most block of the right sub-tree, it will be reverse with respect to the previous block which is otherwise obtained from prefix-sum as they differ by 1.

Since all the other blocks of the right sub-tree are "consistent" with respect to the left-most block of the right sub-tree, if the left-most block has it value changed from odd to even or even to odd by adding one to it, the other blocks of the right sub-tree also make a similar change as that 1 is propagated by the prefix-sum so that the "consistency" is preserved with respect to the left-most block.

In case the two sub-tree are connected by a dashed edge, the direction is the same for the left-most block of the right sub-tree and the right-most block of the left sub-tree. Since in the prefix-sum there is no difference in value between them, both of them will be either odd or even. If the direction of the other blocks needed to be changed for this purpose to preserve consistency, this change is propagated by the prefix-sum.

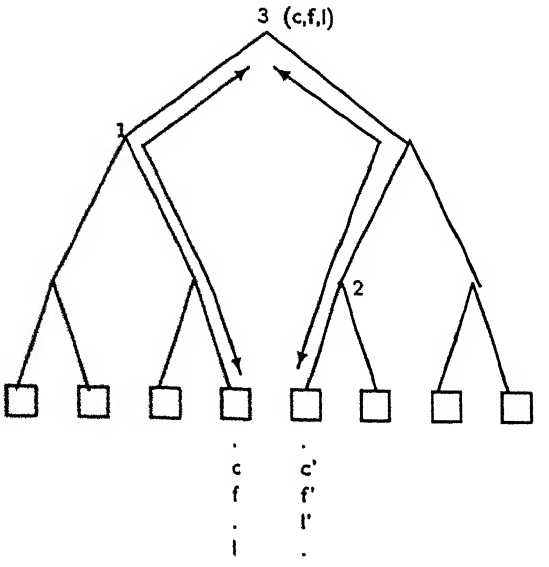


Figure 4.4:

Hence the theorem is true for height=$n+1$. ∎

Thus the merger of blocks in a Tree Machine can be achieved as described above.

4.2 Obtaining final order on Tree Machine

In this section, we look at the problem of obtaining the final order on a Tree Machine. We first determine (as described in previous section) whether a block has to be reversed or not; we reverse the block, in case a reversal is required. We still have to merge these *blocks* so that a final order involving all the nets is obtained.

Procesors at each internal node finds the common elements between the blocks of its right and left sub-tree as shown in Figure 4.4. For this, each processor passes messages to its left child and right child enquiring about its common elements. If a left child receives a message from its parent enquiring about the *parent's* common elements then it passes the message to its right child (node numbered 1 in the figure). Otherwise if the enquiry is about common elements of *an ancestor* of its parent then it passes the message to its left child (node numbered 2 in the figure). Similarly if a right child receives a message from its parent enquiring about the *parent's* common elements then it passes

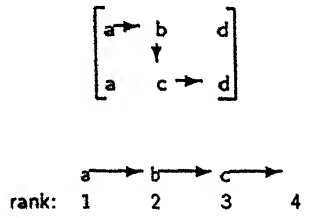


Figure 4.5: Blocks at leaf level with the ranks

the message to its left child. Otherwise if the enquiry is about common elements of an *ancestor* of its parent then it passes the message to its right child.

If a processor receives a message from its child as a reply to an enquiry sent earlier then it passes the message to its parent if it doesn't contain information about its common element. Otherwise if the messages recieved from its children is about its common elements then it compares both the messages to find out the common elements (can be done in constant time as there are only 8 elements). The leaves pass their blocks information in response to an enquiry. Thus each processor knows its common elements between the blocks of its left and right sub-tree.

At the leaves, all blocks have four elements (with a linear ordering defined between them) as shown in Figure 4.5. We rank these elements. These ranks are valid at the leaf-level only. We try to update these ranks at each level of the tree, using a recursive algorithm.

The recursion starts at the root-level. For a node at any level, we recursively update ranks in the left and right sub-tree. It is required to update the ranks at that level also (node numbered 3 in Figure 4.4 under consideration). We thus have two linear orders (ranks) in for both the sub-trees as shown in Figure 4.6. Let us suppose that in both linear orders elements *c,f,l* are common; in figure items of right sub-tree are distinguished by putting a (') on them; e.g., c is for left sub-tree and c' for the right sub-tree. Recall that information about the ranks are at the leaves.

Now at the node numbered 3, we know the common elements between the left and the right sub-tree. This node broadcast a message down the left and the right sub-tree to the leaves enquiring about the ranks of the common elements. Leaves in the left and right sub-trees having the ranks of these elements reply to this message by sending the ranks back.

Then, $lv(x)$, the largest rank in the sub-tree rooted at node x is calculated, using the obvious algorithm in $O(\log n)$ time. Thus, in the figure, $lv(3)$ will be

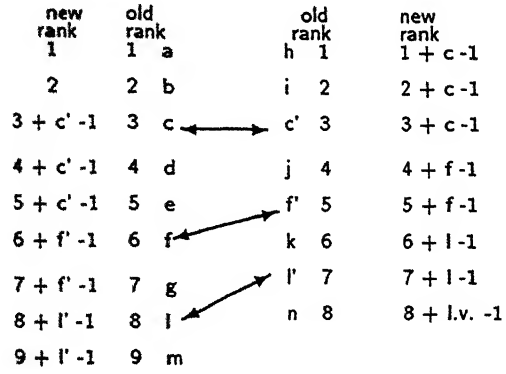


Figure 4.6: The two orders with its rank and updated ranks

known.

Thus the node at which merger should take place (node numbered 3 here) knows about the ranks of the common elements between the left and right sub-tree as well as the largest value of the ranks in the left sub-tree. The node under consideration (node 3 here) passes all these information to all leaves of the tree rooted at that node. The message for the left sub-tree has the bit 0 appended (suffixed) at the end and that for the right sub-tree has the bit 1 suffixed.

As soon as the message arrive at a leaf node, the processor at that node finds out whether it is a leaf of the left or the right sub-treee by checking the last bit.

We describe the procedure for 3 common element case. The procedure for $0, 1$ and 2 common element case is similar (in fact simpler). First the procedure for updating the rank of a leaf x in the left subtree.

Case 1 If the rank is less than the rank of the first common element of the left sub-tree then leave as it is. Thus,

$$if\ Old\_Rank(x) < Old\_Left\_Rank(c)$$

$$then\ Rank(x) = Old\_Rank(x)$$

Case 2 If the rank is less than the rank of the second common element of the left sub-tree but greater than or equal to the rank of the first common element of the left sub-tree then update the rank by adding the rank of the first common elements of the right sub-tree and subtract 1 from it. Thus,

$$if\ Old\_Rank(x) \geq Old\_Left\_Rank(c)$$

$$and\ Old\_Rank(x) < Old\_Left\_Rank(f)$$

$$then$$

$$Rank(x) = Old\_Rank(x) + Old\_Right\_Rank(c) - 1$$

Case 3 If the rank is less than the rank of the third common element of the left sub-tree but greater than or equal to the rank of the second common element of the left sub-tree then update the rank by adding the rank of the second common elements of the right sub-tree and subtract 1 from it. Thus,

$$if\ Old\_Rank(x) \geq Old\_Left\_Rank(f)$$

$$and\ Old\_Rank(x) < Old\_Left\_Rank(l)$$

$$then$$

$$Rank(x) = Old\_Rank(x) + Old\_Right\_Rank(f) - 1$$

Case 4 If the rank is greater than or equal to the rank of the third common element of the left sub-tree then update the rank by adding the rank of the third common elements of the right sub-tree and subtract 1 from it. Thus,

$$if\ Old\_Rank(x) \geq Old\_Left\_Rank(l)$$

$$then\ Rank(x) = Old\_Rank(x) + Old\_Right\_Rank(l) - 1$$

Coming to procedure for updating rank of x in right-subtree.

Case 1 If the rank is less or equal to the rank of the first common element of the right sub-tree then update the rank by adding the rank of the first common element of the left sub-tree and subtract 1 from it. Thus,

$$if\ Old\_Rank(x) \leq Old\_Right\_Rank(c)$$

$$then\ Rank(x) = Old\_Rank(x) + Old\_Left\_Rank(c) - 1$$

Case 2 If the rank is less than or equal to the rank of the second common element of the right sub-tree but greater than the rank of the first common element of the right sub-tree then update the rank by adding the rank of

the second common elements of the left sub-tree and subtract 1 from it. Thus,

$$if\ Old\_Rank(x) \leq Old\_Right\_Rank(f)$$

$$and\ Old\_Rank(x) > Old\_Right\_Rank(c)$$

$$then$$

$$Rank(x) = Old\_Rank(x) + Old\_Left\_Rank(f) - 1$$

Case 3 If the rank is less than or equal to the rank of the third common element of the right sub-tree but greater than the rank of the second common element of the right sub-tree then update the rank by adding the rank of the third common elements of the left sub-tree and subtract 1 from it. Thus,

$$if\ Old\_Rank(x) \leq Old\_Right\_Rank(l)$$

$$and\ Old\_Rank(x) > Old\_Right\_Rank(f)$$

$$then$$

$$Rank(x) = Old\_Rank(x) + Old\_Left\_Rank(l) - 1$$

Case 4 If the rank is greater than rank of the third common element of the right sub-tree then update the rank by adding $lv(LeftRoot)$ (largest rank in the left sub-tree) and subtract 1 from it.

$$if\ Old\_Rank(x) > Old\_Right\_Rank(l)$$

$$then\ Rank(x) = Old\_Rank(x) + lv(3) - 1$$

When the recursion ends at level 0 (the root), all ranks are updated, (at level 0) and hence the ranks at the leaf nodes are the final ranks. Thus the track to which a net belongs can be decided.

As merging at each level takes $O(\log n)$ time and as there are $\log n$ levels, the total time is $O(\log^2 n)$. Less informally, the recurrence relation is

$$T(n) = T(\frac{n}{2}) + O(\log n)$$

$$Or,\ T(n) = O(\log^2 n)$$

Chapter 5

On Heuristics for Single Row Routing Problem

5.1 Motivation for heuristics

In this chapter, we discuss parallel heuristics for Single Row Routing Problem. In [Tsukiyama 82], it was shown that the problem is NP-complete. Hence, to solve the problem in reasonable time, heuristics have to be used. A heuristic for single row routing problem based on the approach introduced in [Kuh 79] was proposed by Tarng et.al.[Tarng 84]. In this chapter, we

a) propose a parallel heuristic similar to that of [Tarng 84] and experimentally compare it.

b) discuss implementation of heuristic of [Tarng 84] on a parallel computer.

We present a new $O(\log n)$ parallel heuristic. Experimentally the proposed heuristic is comparable with the heuristic suggested by Tarng et. al.[Tarng 84].

We also show that the heuristic of Tarng et.al. [Tarng 84] can be implemented in $O(t \log n)$ time on an abstract model of parallel computer.

The heuristic of Tarng et.al. [Tarng 84] is described in Section 5.2. In Section 5.3, an $O(\log n)$ time parallel heuristic based on Interval Graph Colouring Algorithm of Das et.al.[Das 92] is presented. In Section 5.4, an $O(t \log n)$ time parallel algorithm based on Maximum Weighted Independent Set Algorithm due to Bertossi et.al. [Bertossi 87] is described. The modifications as suggested by Du et al. [Du,Liu 87] are discussed in Appendix. Some conclusions are offered in Section 5.5.

5.2 The Single-Row Routing Heuristic of Tarng et al.

The Single-row Routing Heuristic of Tarng et al. is as follows:

```
1   L := Calculate_zone_of_each_net();
2     { returns the set of all the nets such that each net knows the zone it
3       belongs to, the zone being Z_0, Z_1, ..., Z_max_zone_number }
4   L1 := φ;
5   for i := 0 to max_zone_number do
6     { L1 := L1 ∪ Z_i ;
7     for each net n_i ∈ Z_i do
8       Calculate_internal_cut_number(n_i, L1);
9       Calculate_residual_cut_number(n_i, L1);}
10  R := Sort_nets_in_descending_order(L,residual_cut_number);
11  I := Sort_nets_in_descending_order(R,internal cut number);
12  Z := Sort_net_in_ascending_order(I,zone number);
13  for each net n_i ∈ Z do
14    {taken in the order in which they appear in Z }
15    Assign n_i to available track T_x such that | x | is minimum;
```

Basic idea behind the algorithm is to order the nets based on the Zone number, the Internal Cut number and the Residual Cut number such that nets with smaller Zone number come first, within which the nets with higher internal and residual cut number come earlier; the order being arbitrary amongst the nets having the same Zone number, internal cut number and residual cut number. Then we take one net at a time (in order) and try to assign it to the available track T_x such that $| x |$ is minimum.

Figure 5.1 shows the ordering for the nets of Figure 1.2 based on the quantity $[-Z_i, iq_i, rq_i]$ of each net N_i. Here Z_i is the Zone number, iq_i the internal cut number and rq_i the residual cut number.

Let us discuss the implementation of the algorithm on a parallel computer. With $O(n^2)$ processors, the zone number, the internal cut number and residual cut number of the nets can be computed as follows: We assign $(n-1)$ processors to each net N_j. The job of each of $(n-1)$ processors is to see for each i, $1 \leq i \leq n, i \neq j$, whether net N_i overlaps with N_j. Total number of nets overlapping with N_i will give us the $cut - number$ of N_i. This can be

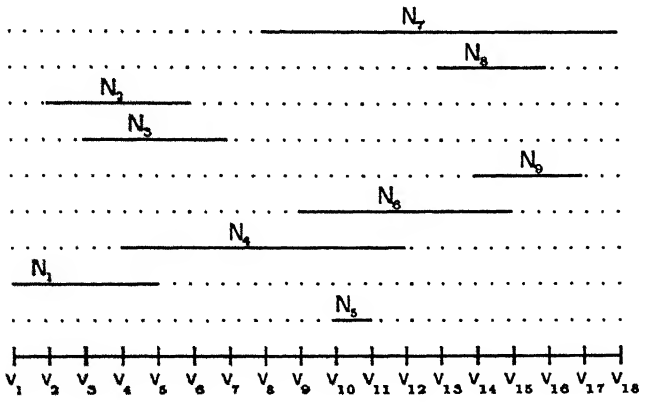


Figure 5.1: Linear ordering of nets

obtained by the usual algorithm of adding n numbers in $O(log\ n)$ time with $\frac{n}{log\ n}$ processors. Hence, each net can find its *Zone number.*

If the Zone number of a net is i, then nets belonging to zone $Z_j, j \leq i$, forms the set $L1$ for that net. Remaining nets form the set $L2$ for that net. We calculate the number of nets in $L1$ and $L2$ over-lapping with that net to get its *internal cut number* and *residual cut number*, respectively.

The nets are, then arranged in a linear order. This can be done in $O(log\ n)$ time using n processors, by sorting using the algorithm of [Cole 88]. The nets are first sorted in descending order on the residual cut number, then again in descending order on the internal cut number and finally, in ascending order on the zone number.

However, the main bottleneck is track assignment as specified in lines 13 to 15. It would require $O(nt)$ time, where t is number of tracks ultimately used and n is the number of nets. In parallel approaches presented in next two Sections, we shall concentrate on this part of the algorithm.

5.3 The O(log n) Heuristic

Since the ultimate objective is to minimise the street congestion in both the streets, minimisation of the number of tracks as a whole would be a healthy heuristic. Based on this idea, we try to assign nets to different track such that the number of tracks is minimum.

The $O(\log n)$ Minimum Colouring algorithm with minimum number of colours of interval graph due to Das et.al.[Das 92] is used here. The interval graph is formed by nodes representing nets and edges connecting the nodes whenever

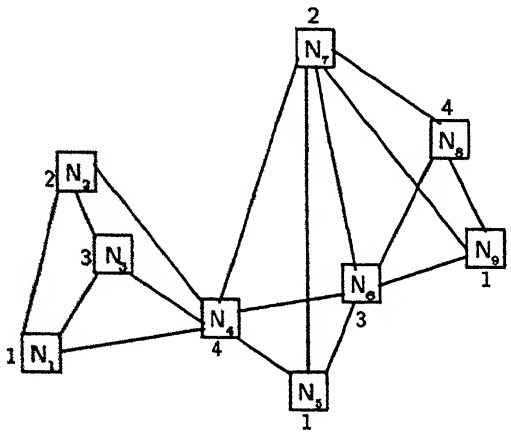


Figure 5.2: Interval Graph of nets along with colour of nodes

the nets represented by the nodes are overlapping. All edges of graph can be found in $O(1)$ time. In the Colouring algorithm, nodes can be of the same colour if they are not connected by an edge, hence mutually non-overlapping. Thus, if the nets of the same colour are assigned to the same track, there won't be any conflict (as they are mutually non-overlapping).

Figure 5.2 shows the interval graph of Figure 5.1 with colours of different nodes.

The linear order amongst various colour classes (the sets of nets, belonging to the same track) is obtained by assigning rank to the various colour classes (sets) as the minimum of the ranks of the nets, belonging to that set (colour class); the rank of the nets being defined by the lexicographic ordering on the quantity $[-Z_i, iq_i, rq_i]$ of the nets.

The set of minimum rank is assigned to track 0, the successive sets of increasing rank to tracks 1, -1, 2, -2 and so on. This would give us the Interval Graphical Representation of the realization.

From Figure 5.2, we find that N_1, N_5 and N_9 will be in the first track i.e. in track T_0; N_4 and N_8 will be in the second track i.e. T_1; N_3 and N_6 will be in the third track, i.e. T_{-1} and finally, N_2 and N_7 will be in the fourth track i.e. T_2.

Table 5.1 shows the minimum tracks obtained when we compared the Tarng et al. heuristic with that of the proposed new heuristic. Similar comparisons when done on randomly generated inputs measured 0.995 by the Student's-T test which can be taken to be good measure of confidence for the proposed new heuristic.

Sl.no	Reference	Tarng et al. heuristic's Street Congestions	New heuristic's Street Congestions
1	[Tarng 84]	2	2
2	[Tarng 84]	2	2
3	[Ting 76]	2	3
4	[Ting 76]	2	2
5	[Ting 76]	2	2
6	[Ting 76]	2	2
7	[Ting 76]	3	3
8	[Kuh 79]	2	2
9	[Kuh 79]	3	3
10	[Kuh 79]	4	4
11	[Kuh 79]	3	4
12	[Du,Liu 87]	3	3
13	[Du,Liu 87]	3	3
14	[Du,Liu 87]	3	3
15	[Tsukiyama 80]	2	2
16	[Tsukiyama 80]	2	2
17	[Tsukiyama 80]	3	3
18	[Han,Sahni 84]	3	3
19	[Han,Sahni 84]	3	3
20	[Du 87]	3	3
21	[Du 87]	3	3
22	[Du 87]	4	4
23	[Du 87]	3	3

Table 5.1: Comparisons of the required number of tracks as given by the Tarng et al. Heuristic and the proposed heuristic

5.4 The O(t log n) Heuristic

The second approach is based on the Maximum Weighted Independent Set Algorithm due to Bertossi et. al. [Bertossi 87]. If weights are assigned to the nets $N_1, N_2, \ldots, N_{i-1}, N_i, N_{i+1}, \ldots, N_n$, (which are assumed to be arranged lexicographically on $[-Z_i, iq_i, rq_i]$), such that weight of the net N_i is greater than the sum of the weights of the nets following it, i.e. $\Sigma_{j=n+1}^{n}$ Weight(N_j) in the interval graph of the nets, then the nets belonging to the Maximum Weighted Independent Set would give us the nets belonging to the first track i.e., T_0.

The nets $n_1, n_2, \ldots, n_i, \ldots, n_{\max}$ belonging to the Maximum Weighted Independent Set are same as the nets $n'_1, n'_2, \ldots, n'_i, \ldots, n'_{\max}$ belonging to first track T_0, as specified by the algorithm, This can be seen as follows. If n_i and n'_i are the first pair of nets which are different, then the weight of n_i is greater than that of n'_i. If this is not the case, that is weight of n'_i is greater than that of n_i, then we would find another Independent Set different from the Maximum Weighted Independent Set such that the sum of the weight of the nets of this set is greater than the Maximum Weighted Independent Set. This is because addition of any amount of nets, whose weights are less than that of n_i, will not make up for the difference in weights between n'_i and n_i.

On the other hand, weight of n_i being greater than that of n'_i would mean that n_i will be selected before n'_i for track assignment in the algorithm. Since there is no over-lapping of n_i with that of $n'_1(= n_1), n'_2(= n_2), \ldots, n'_{i-1}(= n_{i-1}), n_i$ can assigned to the first track before n'_i would be assigned. Hence a contradiction, because according to our assumption n_i wouldn't be assigned to the first track. So n_i must be same as n'_i.

Repeating the above argument for nets $n_j, j > i$, we would find that the nets belonging to the Maximum Weighted Independent Set are the same as those belonging to the first track for above weight assignment.

If weights $1, \frac{1}{k^1}, \frac{1}{k^2}, \ldots, \frac{1}{k^{n-1}}, k > 2$, are assigned to the nets, and if n processors are available then by finding the Maximum Weighted Independent Set, we would find the nets belonging to the first track. Repeating the above process for the remaining nets we would be finding the nets belonging to the second track. Similarly we could find nets of all remaining tracks. Since we have to repeat the Maximum Weighted Independent Set algorithm once for each of the t tracks, and as each iteration of Maximum Weighted Independent Set takes

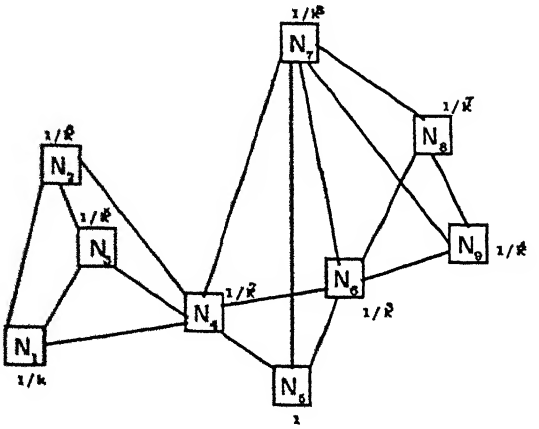


Figure 5.3: Interval Graph for first iteration of O(t log n) time algorithm

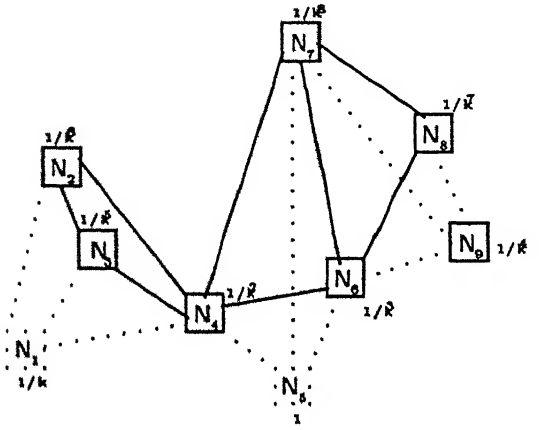


Figure 5.4: Interval Graph for second iteration of O(t log n) time algorithm

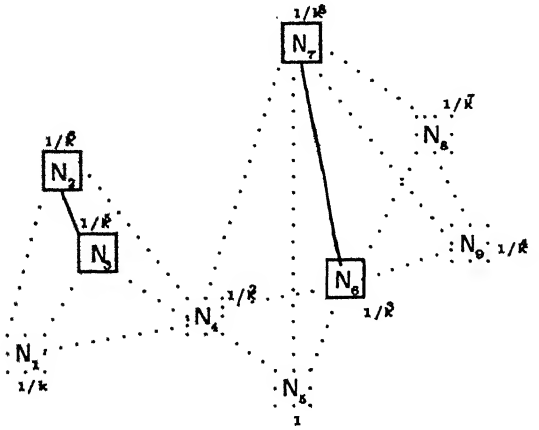


Figure 5.5: Interval Graph for third iteration of O(t log n) time algorithm

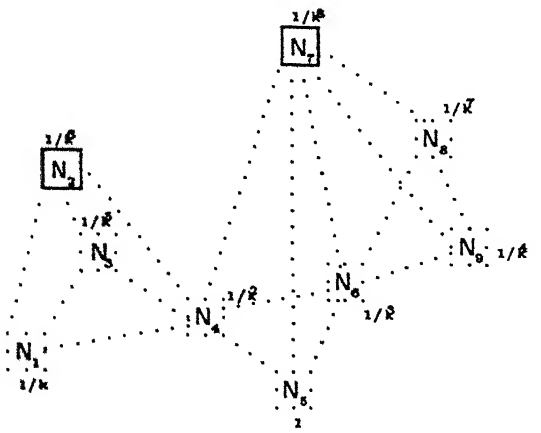


Figure 5.6: Interval Graph for fourth iteration of O(t log n) time algorithm

$O(\log n)$ time, total time complexity will be $O(t \log n)$.

Figure 5.3 shows the Interval Graph along with weights defined for different nodes for the first iteration. N_1, N_5 and N_9 are the nets which form the *Maximum Weighted Independent Set*, hence are in track, T_0.

Figure 5.4 shows the Interval Graph along with weights defined for different nodes for the second iteration. N_4 and N_8 are the nets which form the Maximum Weighted Independent Set, hence are in track, T_1.

Figure 5.5 shows the Interval Graph along with weights defined for different nodes for the third iteration. N_3 and N_6 are the nets which form the Maximum Weighted Independent Set, hence are in track, T_{-1}.

Figure 5.6 shows the Interval Graph along with weights defined for different nodes for the fourth iteration. N_2 and N_7 are the nets which form the Maximum Weighted Independent Set, hence are in track, T_2.

5.5 Final Remarks

The Parallel Heuristic based on Minimum Graph Colouring gives us an $O(\log n)$ algorithm with $O(n^2)$ processors.

Although this is a heuristic, we have found that it managed to find optimum solutions in majority of the examples it was tested with.

The parallel algorithm based on Maximum Weighted Independent Set is more cost effective but is slower; it takes $O(t \log n)$ time with $O(n)$ processors. The necessary modifications in Heuristic (based on ideas of Du and Liu) can be easily incorporated in the proposed parallel heuristics (see Appendix).

Chapter 6

Conclusion and scope for further works

In this thesis, the *Single Row Routing Problem* for some special cases has been solved on parallel models of parallel computers— an abstract model (CREW) and a "practical" architecture, the *Tree Machine*. Some new parallel heuristics have also been proposed.

Scope for further works includes:

1) Reduction of the number of processors as well as developing new heuristics.

2) Generalizing the algorithm for arbitrary (but fixed) prescribed street congestion. This will involve obtaining new necessary and sufficient conditions and, thereby, obtaining the criteria needed to be satisfied by the sequences.

3) Generalizing the algorithm for no cross-over bounds to arbitrary (but fixed) *Cross-over Bound*.

4) Obtaining lower bound on parallel time and processors, for these problems and heuristics. And even designing algorithms which match these lower bounds.

References

[Aho 74] A.Aho,J.Hopcroft, and J.Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley,1974.

[Akl 89] S.G.Akl,The Design and Analysis of Parallel Algorithm,*Prentice-Hall*,1989.

[Albers 92] S.Albers and T. Hagerup, "Improved Parallel Integer Sorting without Concurrent Writing," In *Proc 3rd Ann. ACM-SIAM Symp. on Discrete Alg.*,1992,pp 463-472.

[Alt 86] H.Alt,T.Hagerup,K.Melhorn, and F.Preparata. "Deterministic simulation of idealized parallel computers on more realistic ones". In *Proc. Mathematical Foundations of Computer Science, Lec. Notes in Comp. Science*, Vol. 233. Springer-Verlag, 1986.

[Bertossi 87] A.R.Bertossi and M.A. Bonuccelli, "Some Parallel Algorithms on Interval Graphs", *Discrete Applied Mathematics* vol. 16, 1987, pp. 101-111

[Breuer 72] M.A.Breuer, *Design Automation of Digital Systems,Vol 1: theory and Techniques.* Englewood Cliffs,NJ:Prentice-Hall,1972.

[Cole 88] R.Cole, "Parallel Merge Sort", *SIAM J. Comput.*, vol 17, no. 4, August 1988, pp.770-785

[Das 92] S.K.Das and C.C.Y.Chen,"Efficient Parallel Algorithms on Interval Graph", *Lec. Notes in Comput Sci 605*, pp. 131-143, Parle 1992.

[Du 87] D.H.C. Du, O.Ibarra, and J.F.Naveda. "Single-Row Routing with Crossover Bound," *IEEE Trans. on CAD.* vol. CAD-6,NO.2,March 1987.

[Du,Liu 87] D.H.C.Du and L.C.H.Liu. "Heuristic Algorithms for Single Row Routing", in *IEEE Trans. on Comp.* Vol.C-36,No. 3.March 1987.

[Han,Sahni 84] S.Han and S.Sahni. "Single-Row Routing in Narrow Streets," in *IEEE Trans on Computer-Aided Design*,Vol.CAD-3. No.3,July 1984.

[Hashimoto 71] A.Hashimoto and J.Stevens,"Wire routing by optimizing channel assignment within large apertures," in *Proc.8th Design Automation Conf.*,1971,pp. 155-169.

[Kuh 79] E.S.Kuh,T.Kashiwabana, and T.Fujisawa,"On optimum single-row routing", *IEEE Trans. Circuits Syst.*, vol CAS-26, pp. 361-368, June 1979

[Meijer,Akl 87] H.Meijer and S.G.Akl."Optimal computation of Prefix Sums on a Binary Tree of Processors," *Int. Journal of Parallel Programming* vol.16, no.2,pp.127-136,April 1987.

[Raghavan 83] R.Raghavan and S.Sahni,"Single Row Routing",*IEEE Trans. on Comp.*, vol. C-32,No.3,pp.209-220,March 1983

[Saxena 89] S.Saxena and V.C.Prasad, "On Single Row Routing", *IEEE Transactions on Circuits and Systems*, vol 36, no 7, 1029-1032, July 1989.

[Saxena 94] S.Saxena and N. Malhal Rao,"Parallel Algorithm for finding the K^{th} connected component in an interval graph". *Tech.Rep.* Dept. of Computer Science and Engineering, Indian Institute of Technology,Kanpur,1994.

[So 74] H.C.So,"Some theoretical results on the routing of multilayer printed-wiring boards", *in Proc. 1974 IEEE Int. Symp. Circuits and Systems*, pp. 296-303.

[Tarng 84] T.T.Tarng,M.Marek-Sadowska, and E.S. Kuh, "An Efficient Single-Row Routing Algorithms", *IEEE Trans. on CAD*, vol. CAD-3, No.3, pp. 178-183, July 1984.

[Ting 76] B.S.Ting, E.S.Kuh, and I.Shirakawa, "The multiple routing problem: Algorithms and necessary and sufficient conditions for the single-row single-layer case," *IEEE Trans. Circuits Syst.*, vol.CAS-23, pp. 768-778,1976.

[Tsukiyama 80] S.Tsukiyama, E.S.Kuh, and I.Shirakawa."An Algorithm for Single-Row Routing with Prescribed Street Congestions," in *IEEE Trans. on Circuits and Systems.* Vol.CAS-27,N0.9,September 1980.

[Tsukiyama 82] S.Tsukiyama and E.S.Kuh,"Double row planar routing and permutation layout", *Networks*, vol.12, pp. 286-316, 1982.

Appendix A

Modification of heuristics

Du et al. have proved in [Du,Liu 87] that the heuristic due to Tarng et al. [Tarng 84] is guaranteed to give optimal result only when the nets form a single group; the heuristic fails if there are more groups. Here a group is a collection of nets such that all nets in the collection have atleast one node in common. More than one group results in the formation of zones such that optimal realization is possible only when all net in each zone is assigned tracks according to their maximum *cut number* in that zone.

A.1 Identification of different groups

Definition The *horizontal range* of a net is defined as the range on the node axis from the first node to the last node of that net.

Definition The set of nets that cover at least one common node is said to form a group. Node i is said to be covered by net j if node i lies in between two extreme nodes of a net j.

Definition $Z(i)$ is the set of nets whose horizontal ranges cover node i.

Definition *Zone* is defined to be a set of consecutive nodes on the node axis such that there is a node i in the set and for any other node j in the set, $Z(j) \subset Z(i)$.

Result from [Du,Liu 87] For each zone, the number of elements in $Z(i)$ first increases as i increases, where $i \in$ that zone. After all the nets have appeared, it starts to decrease.

nets	nodes
N1	1,10
N2	2,13
N3	3,5
N4	4,6
N5	7,9
N6	8,16
N7	11,15
N8	12,14

Table A.1: Net List for Example

Therefore, to identify the zones, we find the $Z(i)$ for each i in parallel and then if

(i) $\mid Z(j) \mid > \mid Z(i) \mid and \mid Z(k) \mid > \mid Z(i) \mid$ or,

(ii) $\mid Z(i) \mid = \mid Z(i+1) \mid and \mid Z(i-1) \mid = \mid Z(i+2) \mid and$

$\mid Z(i-1) \mid > \mid Z(i) \mid$

where j and k are the preceeding and succeeding node of i, then i is a point where division of Zone should be made.

For example, consider the nets given in Table A.1. The $Z(i)$ of each nodes is given Table A.2. We observe that the $Z(i)$ first increases till node 4 and then decreases till node 6. From node 7, it again starts to increase till node 8, then again decreases till node 10. From node 11, it starts to increase till node 13 and again to decrease till node 16. Using the result given above, the 1*st* Zone is from 1 to 6, the 2*nd* Zone is from 7 to 10 and the 3*rd* Zone is from 11 to 16.

Next, we compute the q_{ij}'s. Here q_{ij} is the largest value of cut number of nodes of net N_i in the j*th* zone. This can be done in parallel by using $O(\frac{n^2}{\log n})$ processors in $O(\log n)$ time as follows. To every node k in the $j^{th}Zone$, we assign those nets which overlaps with that zone and consider only those portion of the nets which are included in that zone. We find the cut number of node k with respect to these nets only and q_{ij} is assigned the largest value of cut number of all nodes of net N_i. Determination of the net with maximum cut number can be obtained by prefix maximum which can be done in $O(\log n)$ time, with a processor-time product of $O(n)$.

Next, we assign net N_i to the j*th* zone if $q_{ij} = \max_{k=1}^{r} q_{ik}$ where r is the maximum zone number. That is, a net N_i is assigned to zone j if one of the

node	Z(i)	Zone
1	N1	
2	N1,N2	
3	N1,N2,N3	
4	N1,N2,N3,N4	1
5	N1,N2,N3,N4	
6	N1,N2,N4	
7	N1,N2,N5	
8	N1,N2,N5,N6	
9	N1,N2,N5,N6	2
10	N1,N2,N6	
11	N2,N6,N7	
12	N2,N6,N7,N8	
13	N2,N6,N7,N8	3
14	N6,N7,N8	
15	N6,N7	
16	N6	

Table A.2: Table showing the different Zones for the nets in Table A.1

touch point of N_i at which the cut number is largest is in the j^{th} zone. If there are more than one zone in which N_i has the largest cut number, then it is arbitrarily put in any zone.

A.2 Parallelization

Result Given an instance of Single-Row Routing, if all nets in each zone can be assigned to tracks according to their maximum cut numbers in that zone, i.e. nets with largest cut numbers assigned to inner tracks and nets with smaller cut numbers assigned to outer tracks, then the corresponding realization is optimal [Du,Liu 87].

Hence, we should try to apply the parallel heuristic of Section 5.3 or Section 5.4 to each *Zone*. But the problem would be that a single net would be over-lapping with two consecutive Zones and track assigned in both of them might be different.

In sequential heuristic, the track assignment is done for nets in Zone of maximum cut number first (this is the *first track assigned zone*). Then nets in the Zones to the right of this zone are assigned track taking one zone at a time

from left to the right on the node axis (forming the *first series* of Zones which are assigned tracks). Finally, zones to the left of the *first track assigned zone* are assigned tracks taking one zone at a time from right to the left on the node axis (forming the *second series* of Zones which are assigned tracks). Thus we have an ordering of the Zones consisting of the *first track assigned zone* , *first series zones* and *second series zones* in that order.

However, if a net m in the i^{th} zone has larger cut number than net n in the j^{th} zone then m is assigned to a track (in the sequential heuristic) inner than the one to which n is assigned, no matter what may be the relation between i and j in the above mentioned order. Moreover, for nets with same cut number, nets are assigned tracks in the order in which the zone to which they belong appear in the above mentioned order. Thus we have an ordered sequence of nets (the order in which nets are assigned tracks) in the sequential algorithm. This order involves series of nets, each series having nets of same cut number with respect to the zone to which they are assigned. Nets having the largest cut number form the first series, then the nets having the second largest cut number and so on. Within a series the net are ordered according to the order in which the zone to which they belong appear in the order mentioned in the previous paragraph.

To parallelize this step, we build an ordered pair for each net, the first entry is the cut number of the net and the second is "zone number". Here "zone number" is the position of the zone in the linear ordering in which they are assigned tracks in the sequential heuristic (as described in previous paragraph).

The "zone numbers" can be found in parallel as follows: we assign the zone having the maximum cut number first processor. All zones to the right of it are assigned to successive processors. All zone to the left of it are also assigned to successive processors (after the last processor assigned) but here Zones are considered from right to left. Assigning 1 to each processor and finding prefix sums will give us the "zone number".

Having build the ordered pairs, we sort lexicographically on the second number and then on the first number (using algorithm of [Albers 92]). Sorting on the second entry is done in increasing order and on the first entry in the decreasing order.

The sorted pairs will give us the order in which weights are to be assigned to the nets in the parallel heuristics of Section 5.3 or Section 5.4. The parallel heuristics of Section 5.3 or Section 5.4 can now be applied.

i,j	q_{ij}	i,j	q_{ij}	i,j	q_{ij}
N1,1	0	N1,2	2	N1,3	0
N2,1	1	N2,2	0	N2,3	3
N3,1	3	N3,2	0	N3,3	0
N4,1	3	N4,2	0	N4,3	0
N5,1	0	N5,2	3	N5,3	0
N6,1	0	N6,2	3	N6,3	0
N7,1	0	N7,2	0	N7,3	2
N8,1	0	N8,2	0	N8,3	3

Table A.3: q_{ij} of the nets given in Table A.1

For the nets in Table A.1, q_{ij}'s are as shown in Table A.3.

Accordingly,

$(N3, N4) \in Zone1$,

$(N1, N5, N6) \in Zone2$ and

$(N2, N7, N8) \in Zone3$.

Within Zone 1 , $N3$ precedes $N4$.

Within zone 2 , $N5$ precedes $N6$, $N6$ precedes $N1$.

Within zone 3 , $N2$ precedes $N8$, $N8$ precedes $N7$.

Nets in $Zone1$ are first assigned tracks. Then for nets in $Zone2$ and at last, nets in $Zone3$ are assigned tracks. Hence the order is:

{cut-no. 3, Zone 1} $N3, N4$,

{cut-no. 3, Zone 2} $N5, N6$,

{cut-no. 3, Zone 3} $N2, N8$,

{cut-no. 2, Zone 2} $N1$,

{cut-no. 2 ,Zone 3} $N7$.

The interval graphical representation of the nets after the application of parallel heuristic for Tarng et al. heuristic is as shown in Figure A.1.

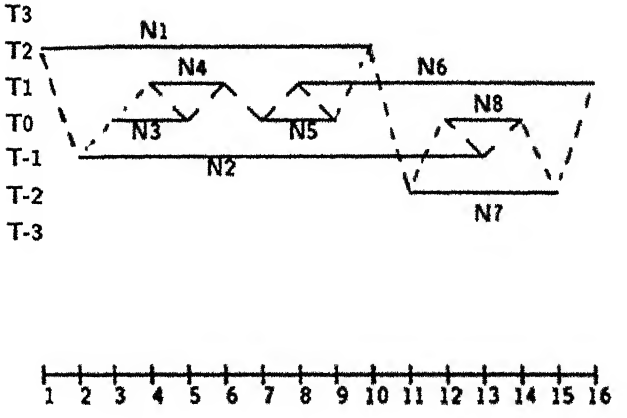


Figure A.1: The Interval Graphical Representation